CS 360: Machine Learning

Prof. Sara Mathieson

Fall 2019

HAVERFORD
COLLEGE

Admin

- **Roster** is semi-finalized

- **First lab today**, may need to bring a computer if you’re on the waitlist and have never taken a CS course at Haverford

- **Lab 1** due Tues night (next office hours Friday 3-5pm) if you did NOT get a Piazza notification about Lab 1, email me ASAP

Outline for Sept 5

- Reading quiz 1
- Introductions
- Style guidelines for Python
- Continue K-nearest neighbors
- Featurization (intro)
- If time: begin entropy and decision trees

Outline for Sept 5

- Reading quiz 1
- Introductions
- Style guidelines for Python
- Continue K-nearest neighbors
- Featurization (intro)
- If time: begin entropy and decision trees

Reading Quiz 1

1) *Generalization*: ability to answer new questions related to the topic studied

2) No! If we look at the test data (either the *features* or the *labels*), then any measurement of the performance of our algorithm becomes inaccurate

3) *Multiclass classification*

4) *Regression*

Outline for Sept 5

- Reading quiz 1
- Introductions
- Style guidelines for Python
- Continue K-nearest neighbors
- Featurization (intro)
- If time: begin entropy and decision trees

Outline for Sept 5

- Reading quiz 1
- Introductions
- Style guidelines for Python
- Continue K-nearest neighbors
- Featurization (intro)
- If time: begin entropy and decision trees

Python style

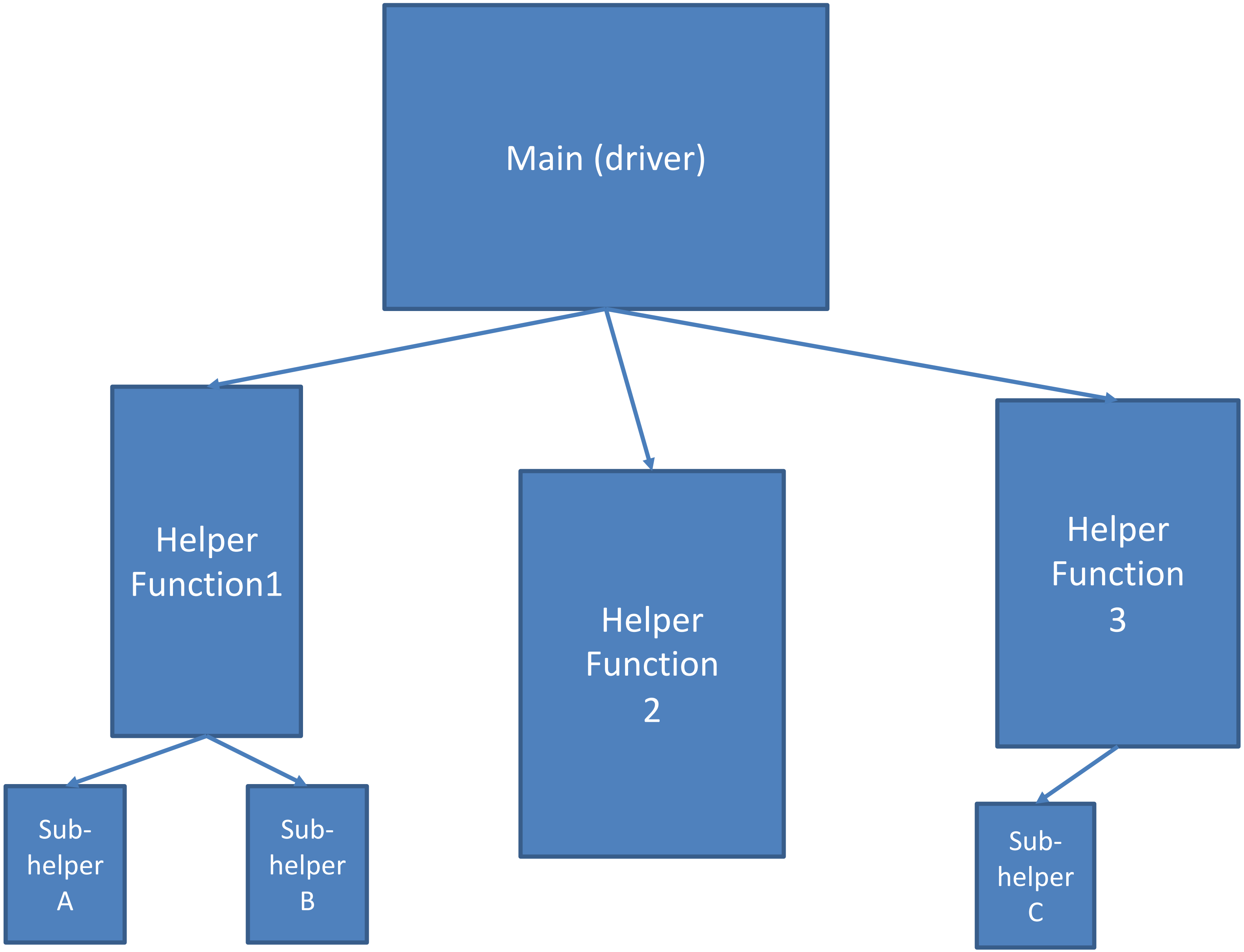
- Decompose code into natural functions
- Avoid global variables (sometimes useful)
- Include a file header with purpose, author, and date
- Include headers for each function
- No lines over 80 chars
- Variable names implicitly show type
- Include line breaks and comments!

Python style examples

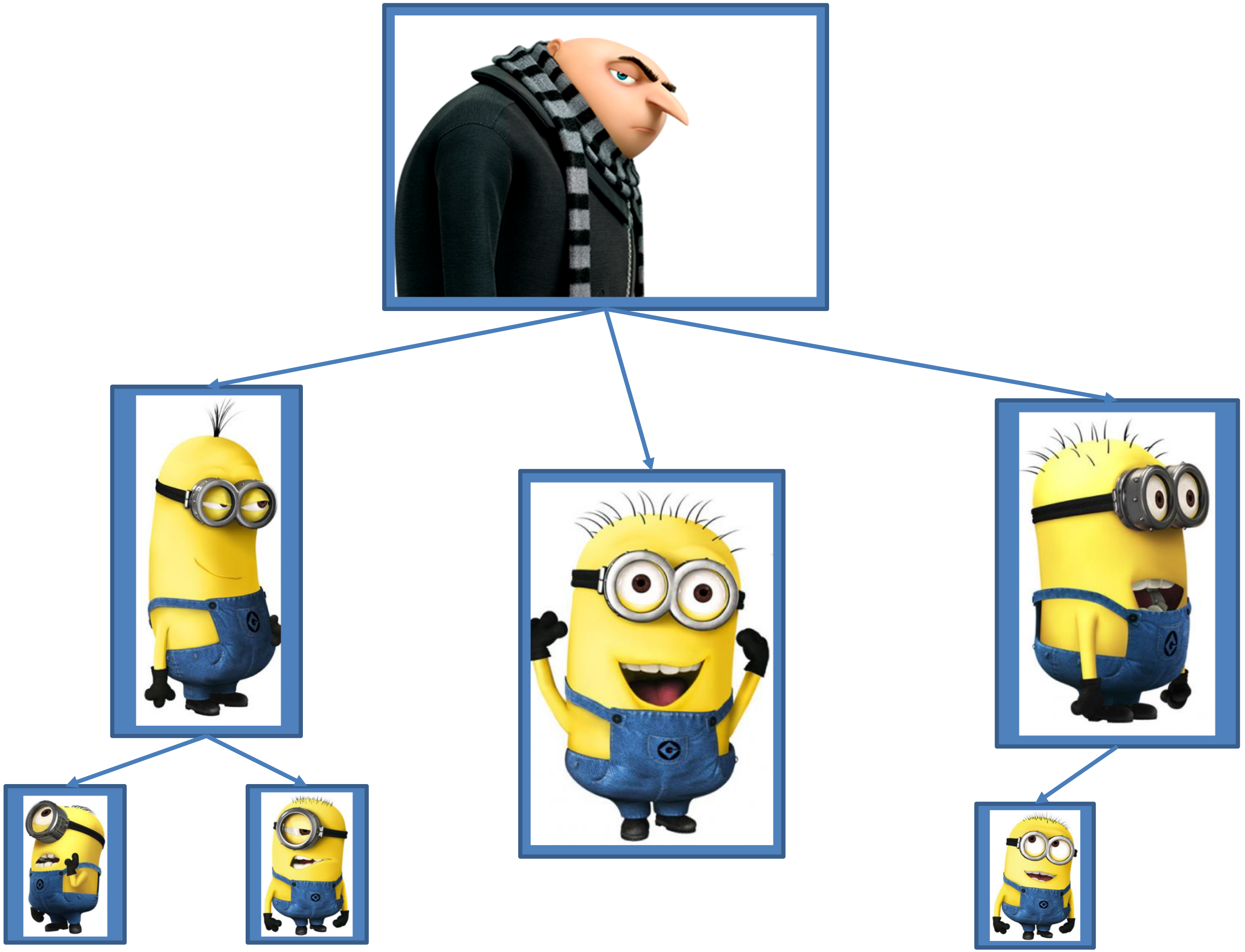
```python
"""
Ask the user for their name and welcome them to CS21.
Author: Sara Mathieson
Date: 9/7/18
"""

def main():

    # ask user for their name and print greeting
    name = input("Enter your name: ")
    print("Hello", name, "!")

main()
```

```python
def factorial(n):
    """
    Given a non-negative integer n, return n! = n*(n-1)*(n-2)....3*2*1.
    """
    fact = 1 # set up an accumulator variable
    for i in range(n):
        fact = fact * (i+1) # accumulator pattern
    return fact
```

Structure of main and "helper" functions



Structure of main and "helper" functions



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".

Reasons to use TDD

- Creates code that is easier to implement, debug, modify, and extend

- Avoids going off in the wrong direction (i.e. implementing functions that are not useful or don't serve the program)

- Creates code that is easier for you or someone else to read and understand later on

Outline for Sept 5

- Reading quiz 1
- Introductions
- Style guidelines for Python
- Continue K-nearest neighbors
- Featurization (intro)
- If time: begin entropy and decision trees

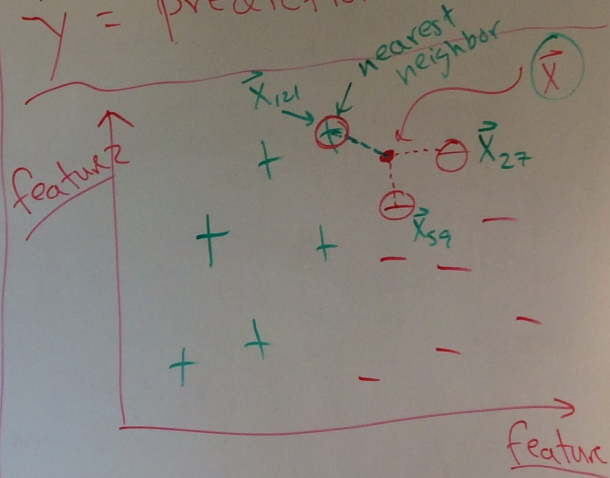
X = $\underbrace{n \times p}_{\text{row} \times \text{col}}$ matrix of features → "rom-com"

X_{train} = training data (80%)

X_{test} = test data (20%)

y = label / output (vector)

$\hat{y}$ = prediction.



$K=1 \Rightarrow$ nearest neighbor

$\hat{y} = \text{sign}(+1) = +1$

$K=2 \Rightarrow \hat{y} = \underbrace{\text{sign}(+1-1)}_{\text{sign}(0) \Rightarrow -1} = -1$

$K=3 \Rightarrow$ 3 nearest neighbors

$\hat{y} = \text{sign}(+1-1-1) = -1$

$$\text{sign}(z) = \begin{cases} +1 & \text{if } z > 0 \\ -1 & \text{if } z \le 0 \end{cases}$$

conservative

Algorithm input: $\vec{x}$ (test), K, X_{train}
(binary) $y \in \{-1, +1\}$ y_{train}

$\vec{S}$ = vector of len n

for $i = 1 \ldots n$:
 $S_i = [d(\vec{x}, \vec{x}_i), i]$

test, train, index

$\boxed{\text{or}}$ y_i

sort $\vec{S}$ by dist.

votes = 0
for $k = 1 \ldots K$:
 [dist, i] = S_k
 votes += y_i
return sign(votes)

Multiclass

$N_K(\vec{x})$ = set of nearest neighbors for $\vec{x}$

$$N_3(\vec{x}) = \{\vec{x}_{121}, \vec{x}_{27}, \vec{x}_{59}\}$$

$$p(y = c \mid \vec{x}) = \frac{1}{K} \sum_{\vec{x}_i \in N_K(\vec{x})} \mathbb{1}(y_i = c)$$

$\{+1, -1\}$

$\{1, 2, 3, 4, 5\}$

$K = 5$

$$p(y = \bullet \mid \vec{x}) = \frac{1}{5}$$

$$p(y = \square \mid \vec{x}) = \frac{2}{5}$$

$$p(y = \Delta \mid \vec{x}) = \frac{2}{5}$$

K-nearest neighbors creates implicit decision boundaries

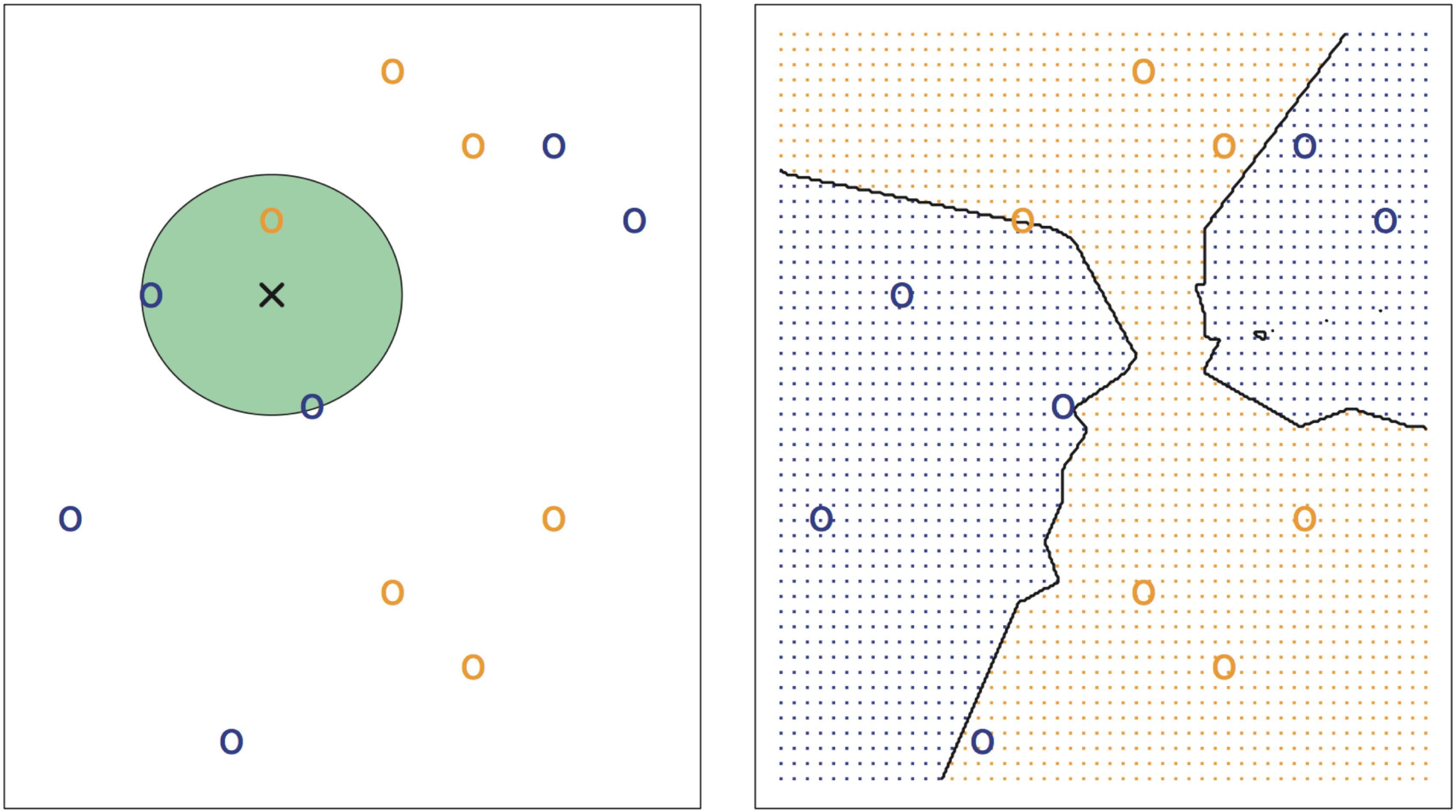


Figure 2.14 from ISL book, KNN with two classes (C=2), and K=3

Outline for Sept 5

- Reading quiz 1
- Introductions
- Style guidelines for Python
- Continue K-nearest neighbors
- Featurization (intro)
- If time: begin entropy and decision trees

Terminology

- *Features:* feature names
  - i.e. shape
- *Feature values:* what values are possible
  - i.e. {circle, square, triangle}
- *Feature vector:* values for a particular example
  - i.e. $\boldsymbol{x} = [x_1, x_2, x_3, \ldots, x_p]$

Terminology

- *Decision boundary*: separates regions of the feature space that would be classified as positive or negative (or multiclass)
- *Underfitting*: “had the opportunity to learn something but didn’t” (Duame)
- *Overfitting*: memorized individual training examples (fit to noise) and can’t generalize

Handout 2
(find and work with a partner)

Comparison of decision boundaries

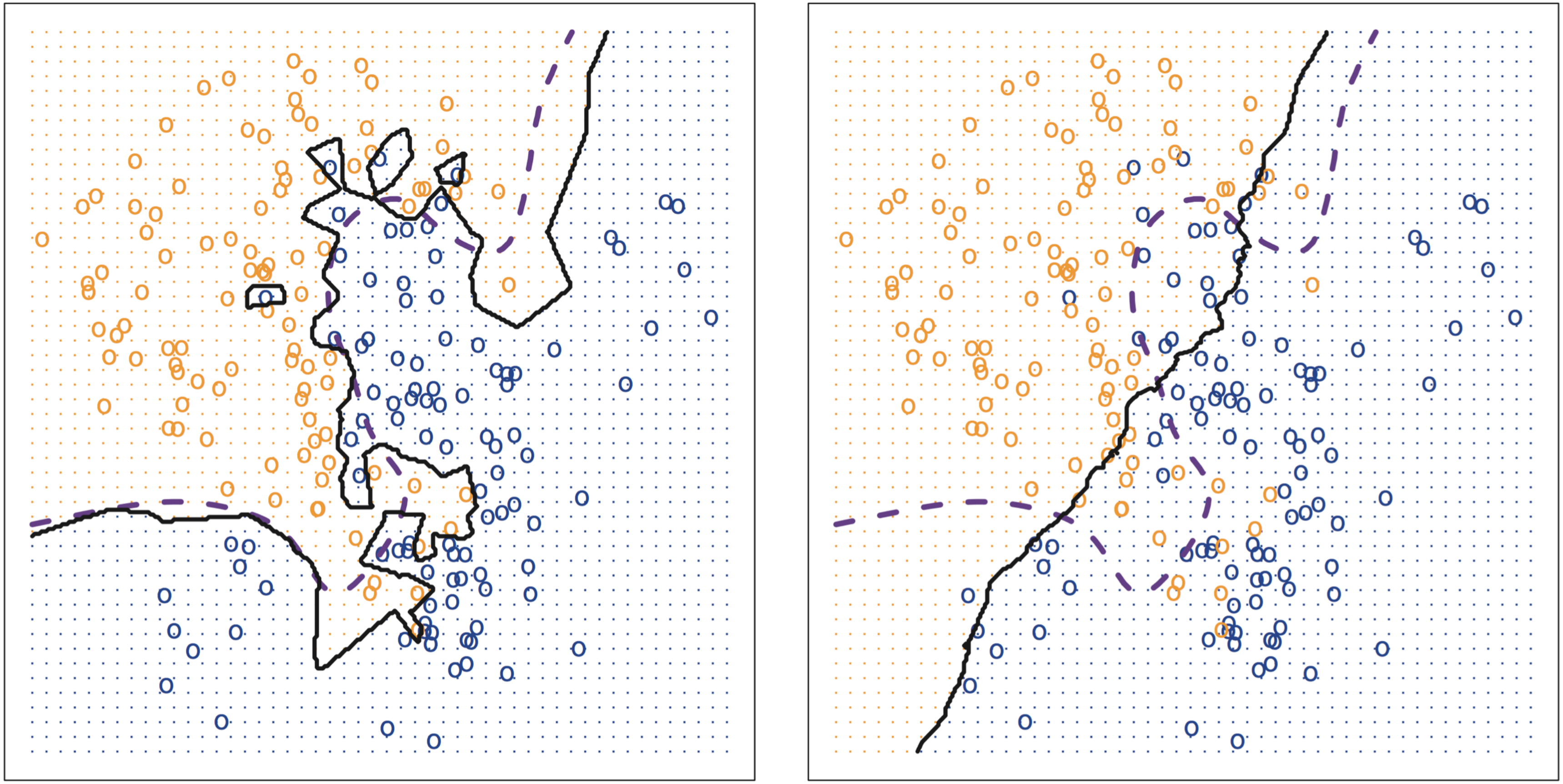


Figure 2.16 from ISL book (dashed line is “ideal” boundary)

Comparison of decision boundaries

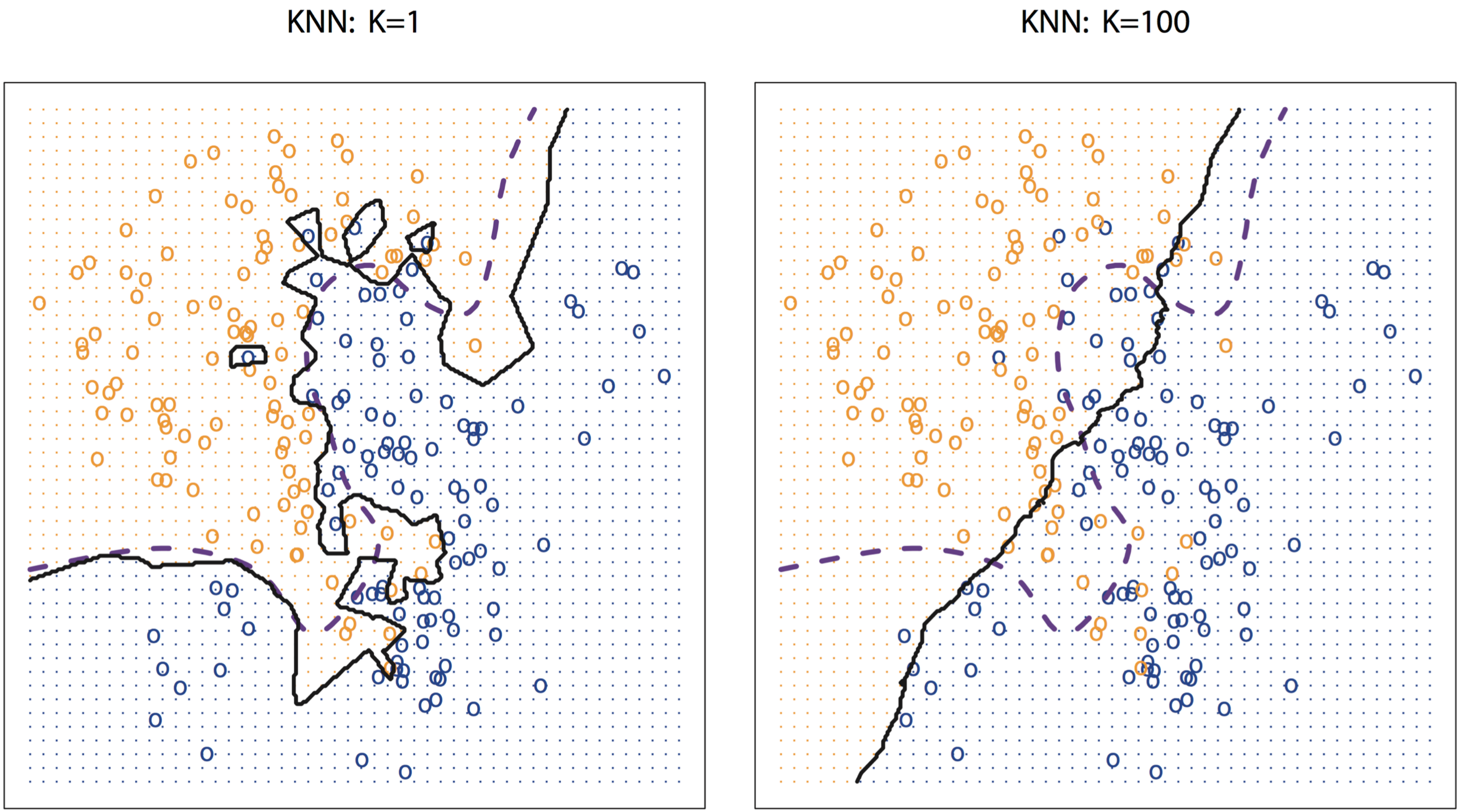
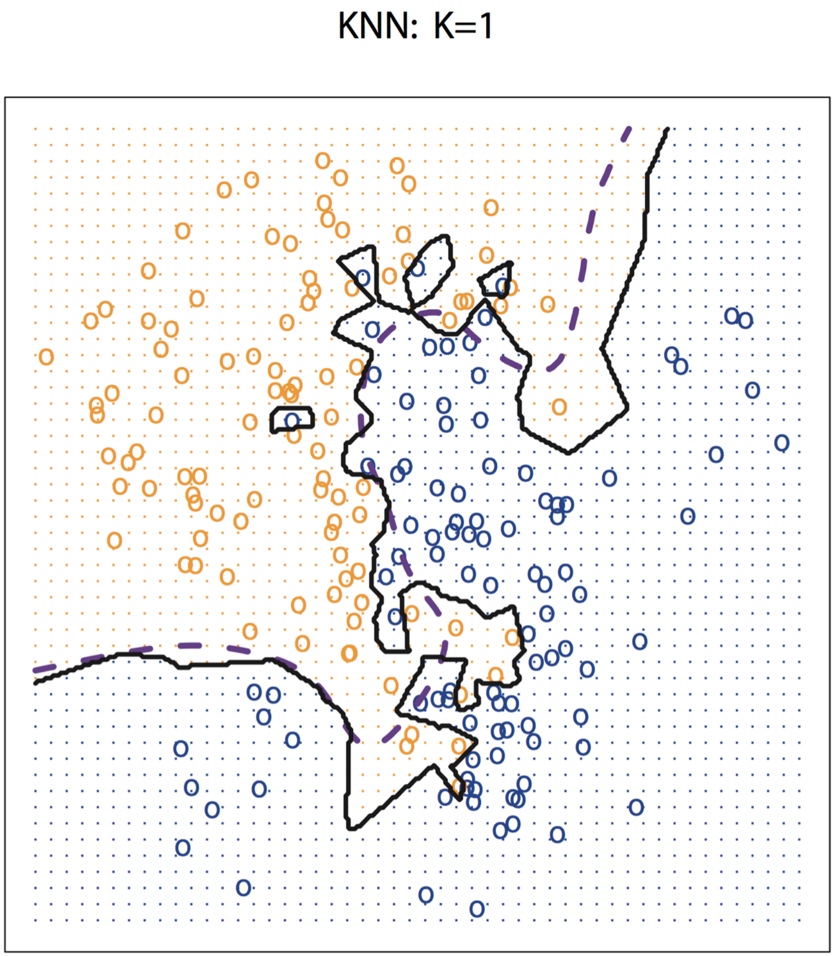
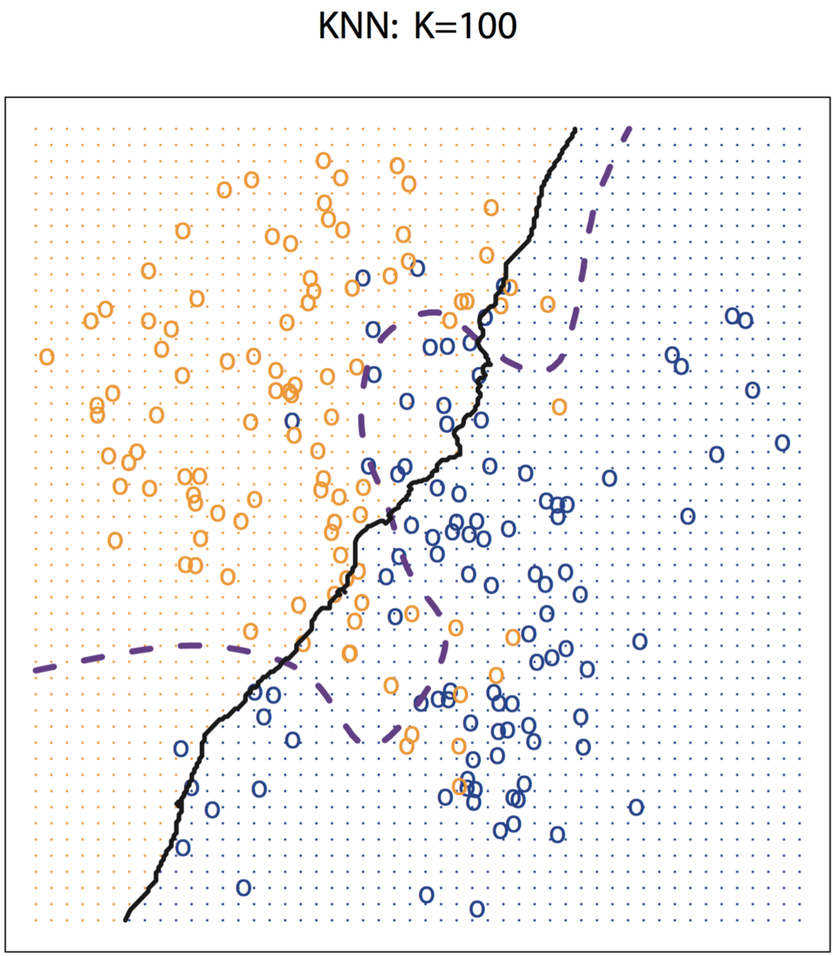


Figure 2.16 from ISL book (dashed line is “ideal” boundary)

Comparison of decision boundaries



Overfitting



Underfitting

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

Featurization (rule of thumb)

Duame, Chap 3

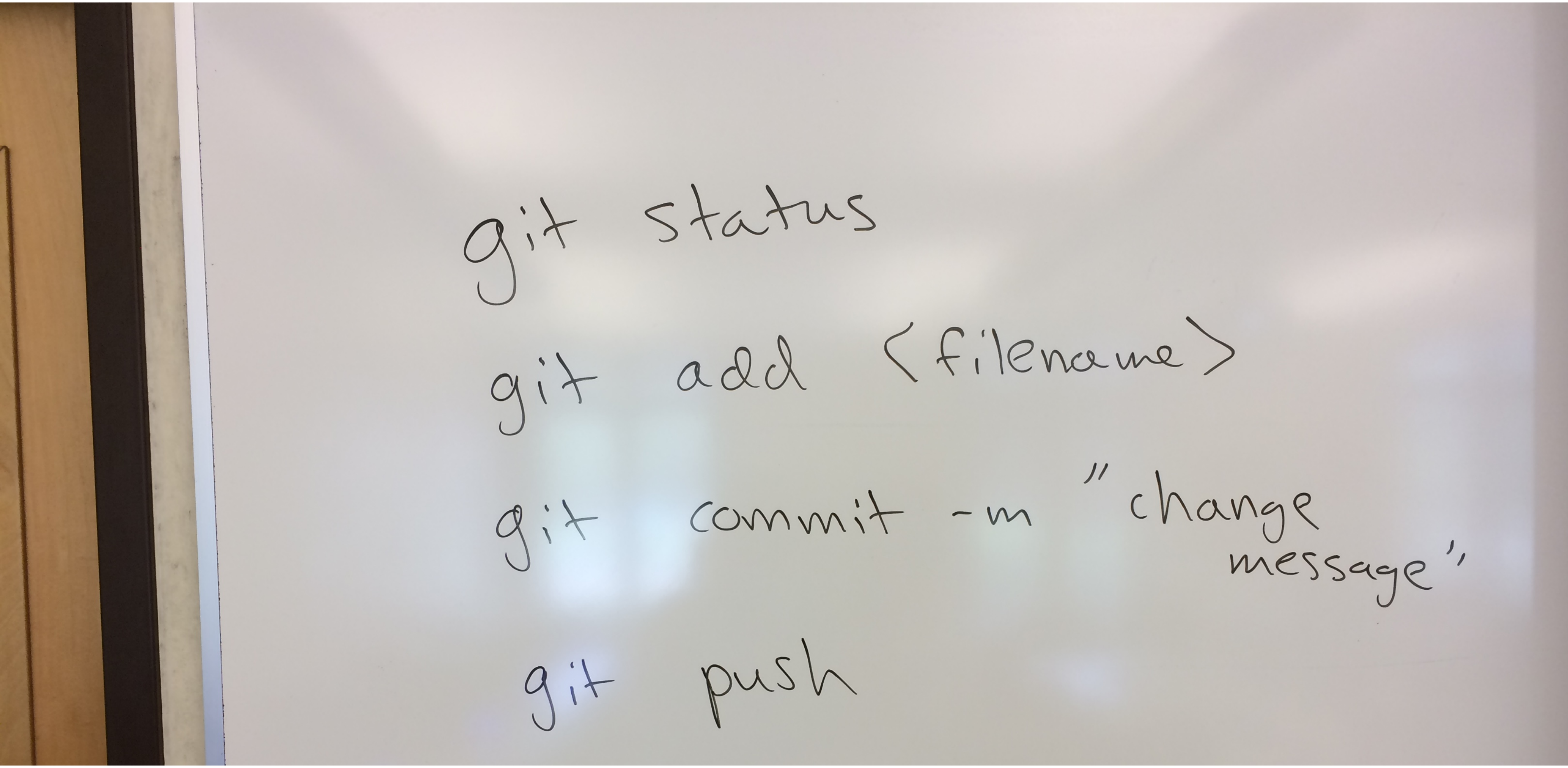
- Real-valued features get copied directly.
- Binary features become 0 (for false) or 1 (for true).
- Categorical features with V possible values get mapped to V-many binary indicator features.

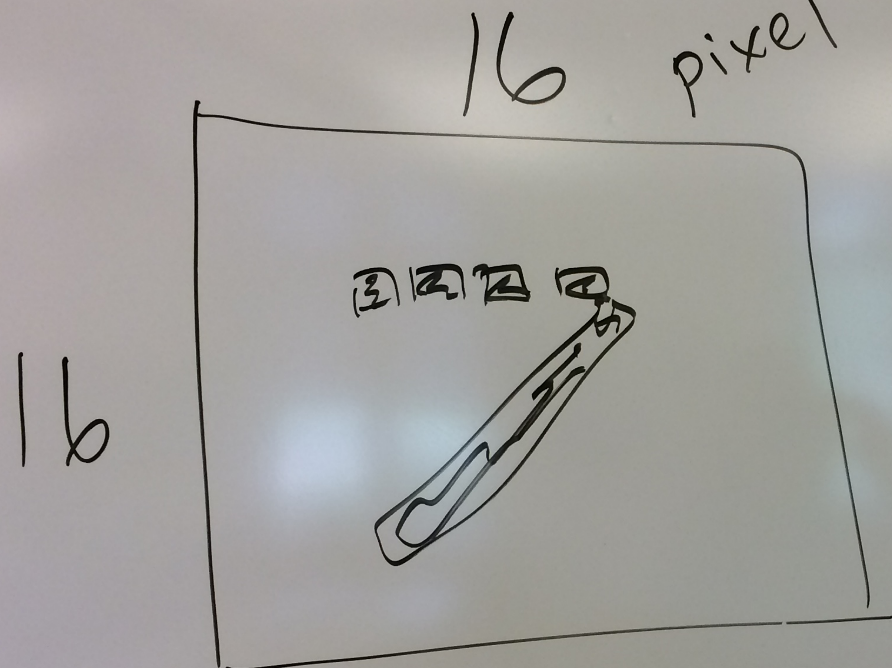
Haven't discussed:

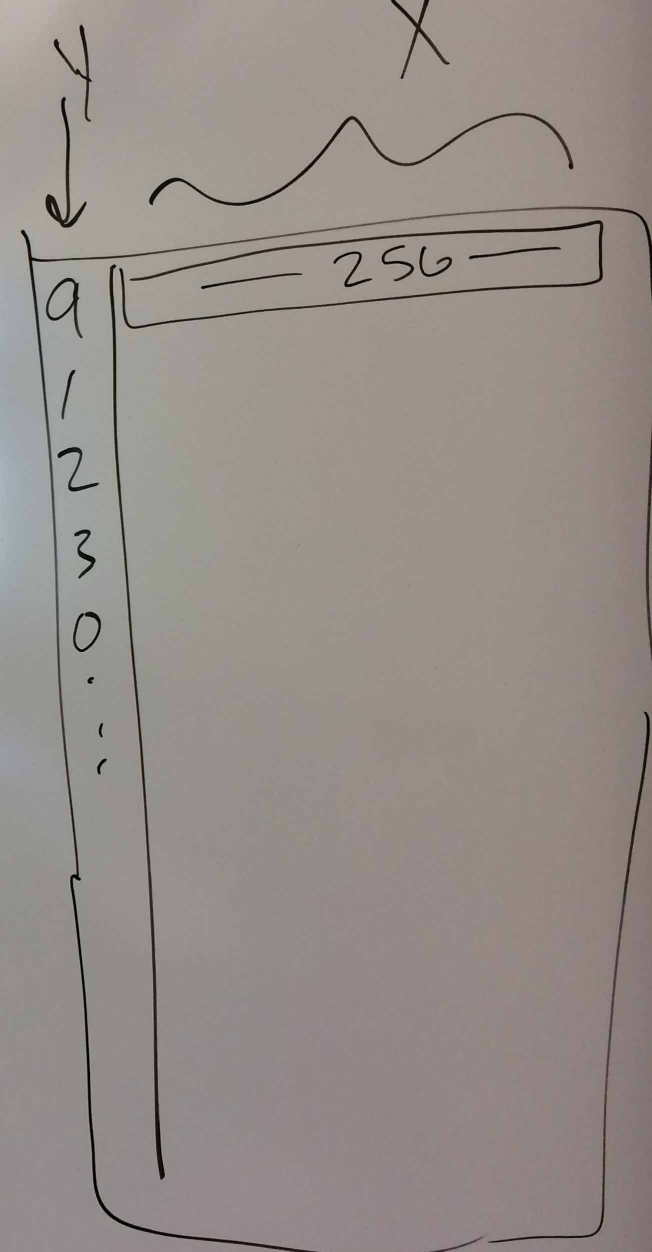
-normalization

-categorical variables on a spectrum

Lab 1 Notes


git status
git add <filename>
git commit -m "change message"
git push


16 pixel
16


y
X
256
9
1
2
3
0