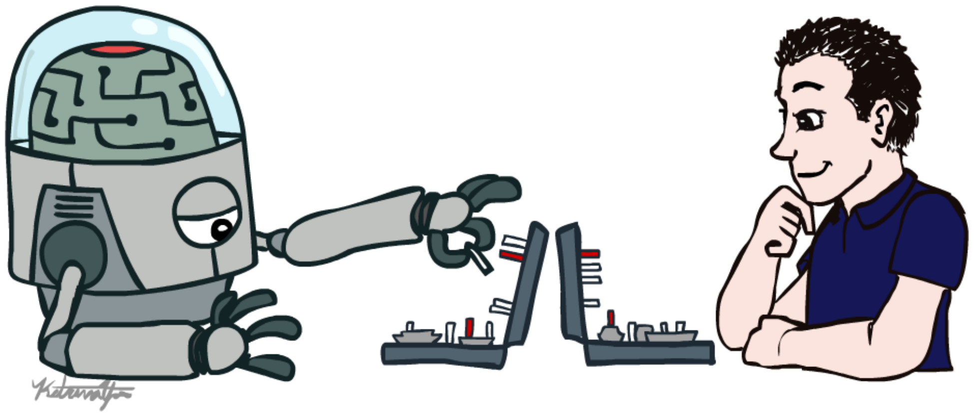
Lecture 15



Ashis Kumar Chanda

chanda@rowan.edu

Rowan University

Practical Issues in ML

Slides adapted from David Kauchak

In theory, there is no difference between theory and practice. But, in practice, there is.

– Jan L.A. van de Snepscheut

Good Features

- ``garbage in, garbage out''
- Consider "creditworthiness" prediction task

$$\vec{x} = (x_1, x_2, \ldots, x_d)$$

- Does it matter if "age" or "salary" is x_1, x_2, or x_d?
- Does learning task change at all?
  - No, order does not matter
  - Implies features can be arbitrarily reordered
  - *Of course, order must be consistent across examples*

- *ML algorithms care about **feature values**, not **features***

Irrelevant and Redundant Features

- An ***irrelevant feature*** is completely uncorrelated with prediction task
  - E.g., word frequency of "the" for movie review sentiment prediction

- Two features are **redundant** if highly correlated
  - regardless of whether they are correlated with the task or not
  - E.g., adjacent pixels are usually the same color

- *How robust is an algorithm to irrelevant and redundant features?*

Bad Features?

- Not useful to consider 999 great features and 1 bad feature
- More common for bad features to outnumber the good features
  - often by a large degree
- What percent of features are “good”?
  - Consider bag of words (or bag of pixels)
  - How many words are *actually useful* for predicting positive and negative movie reviews?
  - Feature vector length could be size of English dictionary (~50k)

Decision Tree + Bad Features

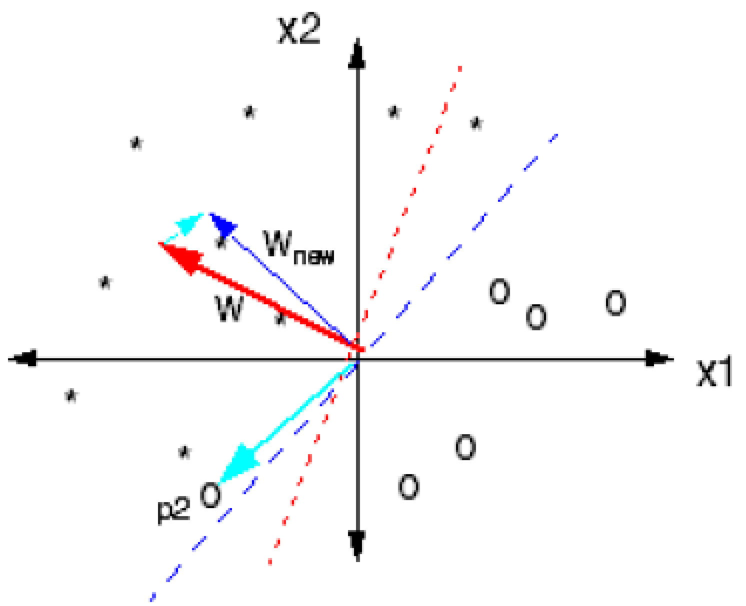
- Shallow decision trees explicitly select features that are highly correlated with the label
- Limiting depth (hopefully) throws away irrelevant features
- What about redundant features?
  - Redundant features are almost certainly thrown out
    - once you select one feature, the second is mostly useless
- Subtle concern with irrelevant features
  - Chance correlations (especially deeper in tree)
  - Need more training data to avoid chance correlations

K-NN + Bad Features

- K-NN is less robust to bad features
  - “all features matter equally”
- Irrelevant features can completely mess up KNN prediction
  - Recall “curse of dimensionality”
  - The distances of random, high-dimensional points tend to converge
- Redundant features exaggerate the effect
  - Can make neighbors appear more similar or more different

Perceptron + Bad Features

- Hopefully, perceptron learns to assign zero weight to irrelevant features
- Consider a binary feature that is randomly one or zero independent of the label
- If the perceptron makes (roughly) equal updates for positive examples and negative examples, there is a reasonable chance this feature weight will be zero (or small)



Feature Pruning

- Consider text categorization
  - Some words simply do not appear very often
  - Are these rare words useful or not useful for classification?
    - Not clear. Rare words might be **salient** or **noise**.
- ***Feature pruning:*** if a feature appears <K times (in training), remove it
  - Also applies to common features (appear all-but-K times) like “the”
- Choice of K depends on data size
  - Text data set with 1000s of documents, K = 5 is reasonable
  - Internet-scale problem K = 50, 100, 200 are reasonable
  - According to Google, the following words appear 200 times on the web: **agaggagctg**, **setgravity**, **rogov**, **piyushtwok**, **nesmysl**, **brighnasa**
  - For comparison, the word ``the'' appears 19 billion times
- For low K, pruning does not hurt (and sometimes helps) but eventually we prune away all the interesting features and performance suffers

Rules of Thumb

Be very careful in domains where:

- the number of features > number of examples
- the number of features ≈ number of examples
- the features are generated automatically
- there is a chance of “random” features

In most of these cases, features should be removed based on some domain knowledge (i.e. problem-specific knowledge)

Feature Scale

Length	Weight	Color	Label
4	4	0	Apple
5	5	1	Apple
7	6	1	Banana
4	3	0	Apple
6	7	1	Banana
5	8	1	Banana
5	6	1	Apple

Length	Weight	Color	Label
40	4	0	Apple
50	5	1	Apple
70	6	1	Banana
40	3	0	Apple
60	7	1	Banana
50	8	1	Banana
50	6	1	Apple

Does it matter if we measure length in cm or mm?

Would our three classifiers (DT, k-NN and perceptron) learn the same models on these two data sets?

Decision Tree?

Length	Weight	Color	Label
4	4	0	Apple
5	5	1	Apple
7	6	1	Banana
4	3	0	Apple
6	7	1	Banana
5	8	1	Banana
5	6	1	Apple

Length	Weight	Color	Label
40	4	0	Apple
50	5	1	Apple
70	6	1	Banana
40	3	0	Apple
60	7	1	Banana
50	8	1	Banana
50	6	1	Apple

Decision trees don't care about scale,
so they'd learn the same tree

K-NN?

Length	Weight	Color	Label
4	4	0	Apple
5	5	1	Apple
7	6	1	Banana
4	3	0	Apple
6	7	1	Banana
5	8	1	Banana
5	6	1	Apple

Length	Weight	Color	Label
40	4	0	Apple
50	5	1	Apple
70	6	1	Banana
40	3	0	Apple
60	7	1	Banana
50	8	1	Banana
50	6	1	Apple

k-NN: NO! The distances are biased based on feature magnitude.

$$D(a,b)=\sqrt{(a_1-b_1)^2+(a_2-b_2)^2+\ldots+(a_n-b_n)^2}$$

K-NN distances

Length	Weight	Label
4	4	Apple
7	5	Apple
5	8	Banana

Which of the two examples are closest to the first?

Length	Weight	Label
40	4	Apple
70	5	Apple
50	8	Banana

$$D(a,b)=\sqrt{(a_1-b_1)^2+(a_2-b_2)^2+\ldots+(a_n-b_n)^2}$$

K-NN distances

Length	Weight	Label
4	4	Apple
7	5	Apple
5	8	Banana

$$D = \sqrt{(7-4)^2 + (5-4)^2} = \sqrt{10}$$

$$D = \sqrt{(5-4)^2 + (8-4)^2} = \sqrt{17}$$

Length	Weight	Label
40	4	Apple
70	5	Apple
50	8	Banana

$$D = \sqrt{(70-40)^2 + (5-4)^2} = \sqrt{901}$$

$$D = \sqrt{(70-50)^2 + (8-4)^2} = \sqrt{416}$$

$$D(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

Perceptron?

Length	Weight	Color	Label
4	4	0	Apple
5	5	1	Apple
7	6	1	Banana
4	3	0	Apple
6	7	1	Banana
5	8	1	Banana
5	6	1	Apple

Length	Weight	Color	Label
40	4	0	Apple
50	5	1	Apple
70	6	1	Banana
40	3	0	Apple
60	7	1	Banana
50	8	1	Banana
50	6	1	Apple

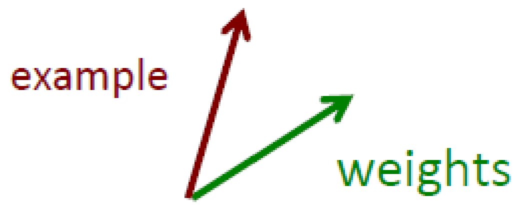
Perceptron: NO!
The classification and weight update are based on the magnitude of the feature value

Geometric view of perceptron update

for each w_i:

$$w_i = w_i + f_i * \text{label}$$

Geometrically, the perceptron update rule is equivalent to “adding” the weight vector and the feature vector

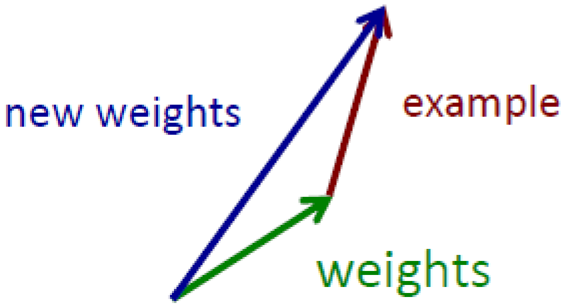


Geometric view of perceptron update

for each w_i:

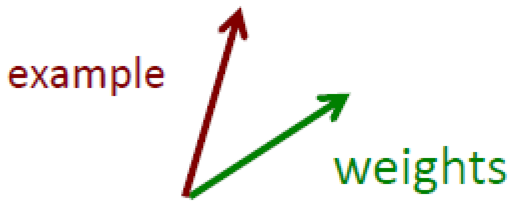
$w_i = w_i + f_i$*label

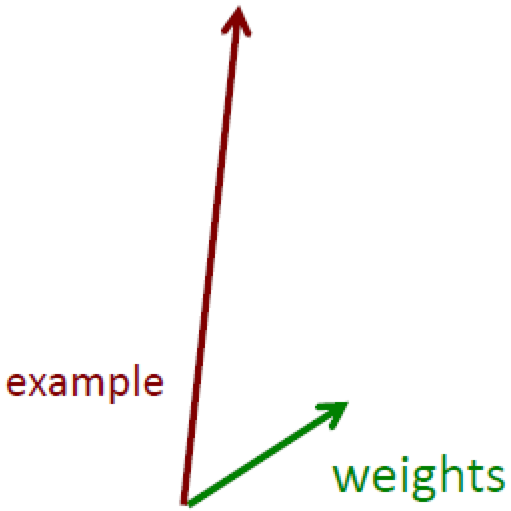
Geometrically, the perceptron update rule is equivalent to "adding" the weight vector and the feature vector



Geometric view of perceptron update

If the features dimensions differ in scale, it can bias the update

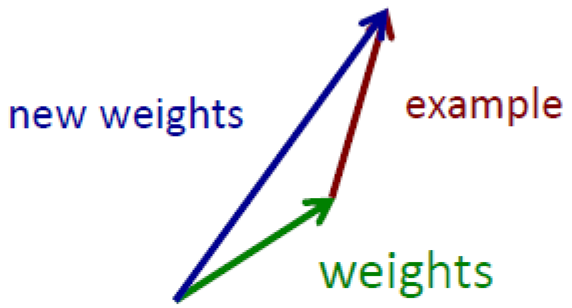


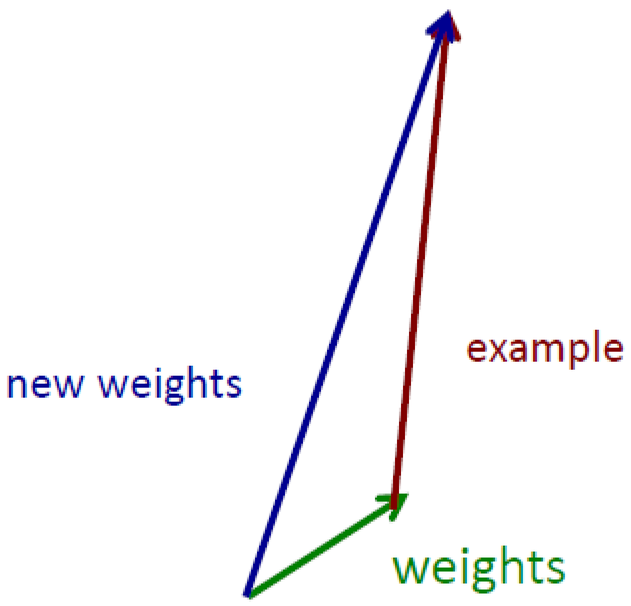


same f1 value, but larger f2

Geometric view of perceptron update

If the features dimensions differ in scale, it can bias the update





- different separating hyperplanes
- the larger dimension becomes much more important

How do we fix this?

Length	Weight	Color	Label
4	4	0	Apple
5	5	1	Apple
7	6	1	Banana
4	3	0	Apple
6	7	1	Banana
5	8	1	Banana
5	6	1	Apple

Length	Weight	Color	Label
40	4	0	Apple
50	5	1	Apple
70	6	1	Banana
40	3	0	Apple
60	7	1	Banana
50	8	1	Banana
50	6	1	Apple

Normalization

- There are two basic types of normalization
  - **feature normalization**: go through each feature and adjust it the same way across all examples
    - Centering: moving the entire data set so that it is centered
    - Scaling: rescaling each feature so that (either):
      - Each feature has variance 1 across the training data.
      - Each feature has maximum absolute value 1 across the training
  - **example normalization**: each example is adjusted individually
    - Scale feature vector so that magnitude is 1

- The goal of normalization is to make it ***easier*** to learn

Normalize each feature

For each feature (over all examples):

Center: adjust the values so that the mean of that feature is 0: subtract the mean from all values

Rescale/adjust feature values to avoid magnitude bias:

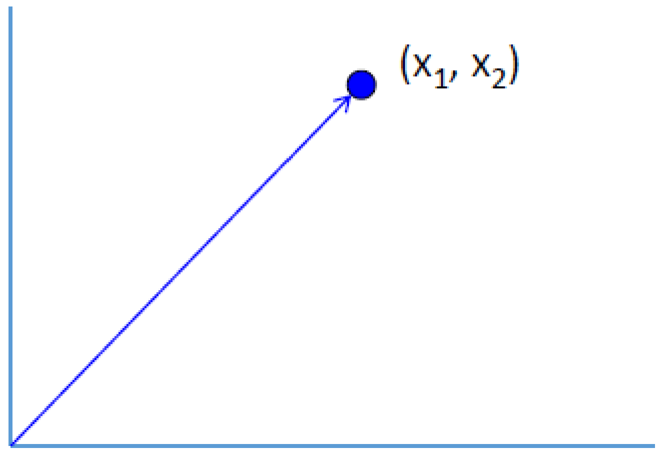
- Variance scaling: divide each value by the std dev
- Absolute scaling: divide each value by the largest value

Pros/cons of either scaling technique?

Example length normalization

Make all examples roughly the same scale, e.g. make all have length = 1

What is the length of this example/vector?



$$length(x) = \|x\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Example length normalization

Make all examples have length = 1

Divide each feature value by ||x||

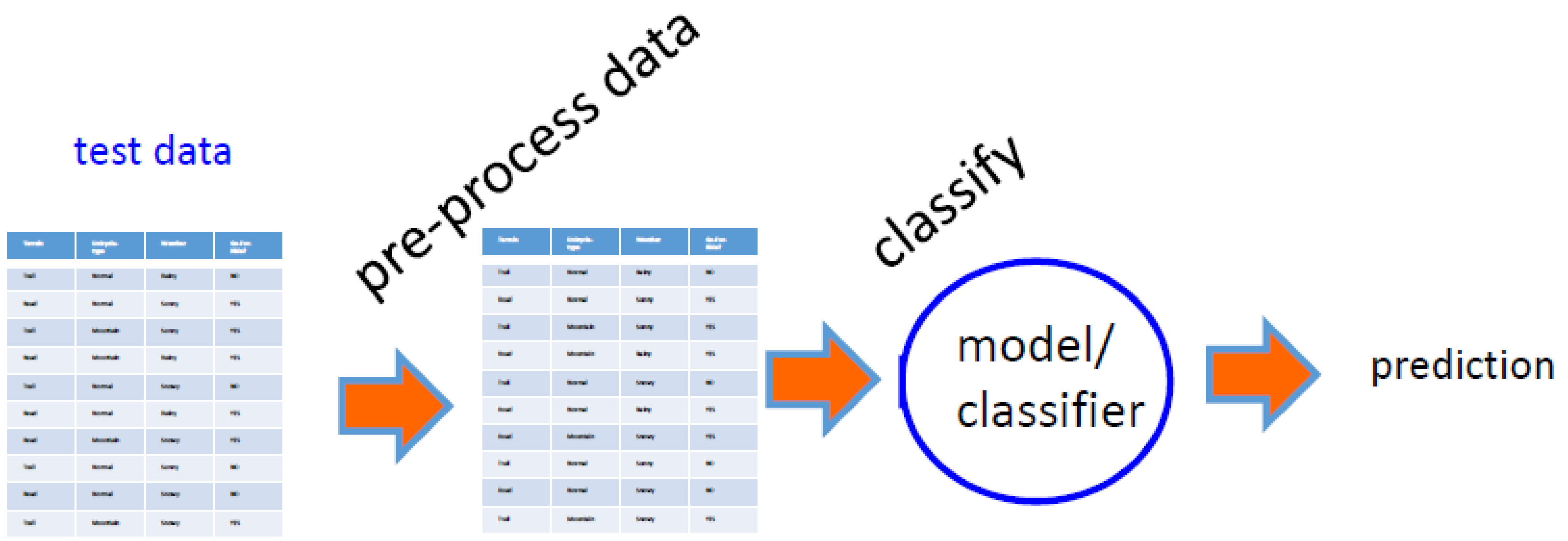
- Prevents a single example from being too impactful
- Equivalent to projecting each example onto a unit sphere

$$length(x) = \|x\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Basic Preprocessing Recipe

- Remove noisy features
- Pick "good" features
- Normalize feature values
  - center data
  - scale data (either variance or absolute)
- Normalize example length
- Finally, train your model!

What about testing?



Whatever you do on training, you have to do the EXACT same on testing!

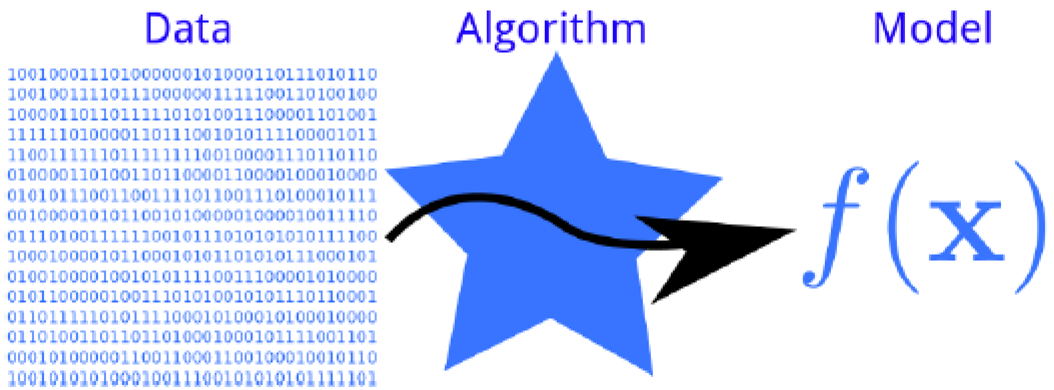
Normalizing test data

- Center: adjust the values so that the mean of that feature is 0: subtract the mean from all values

- Rescale/adjust feature values to avoid magnitude bias:
  - Variance scaling: divide each value by the std dev
  - Absolute scaling: divide each value by the largest value

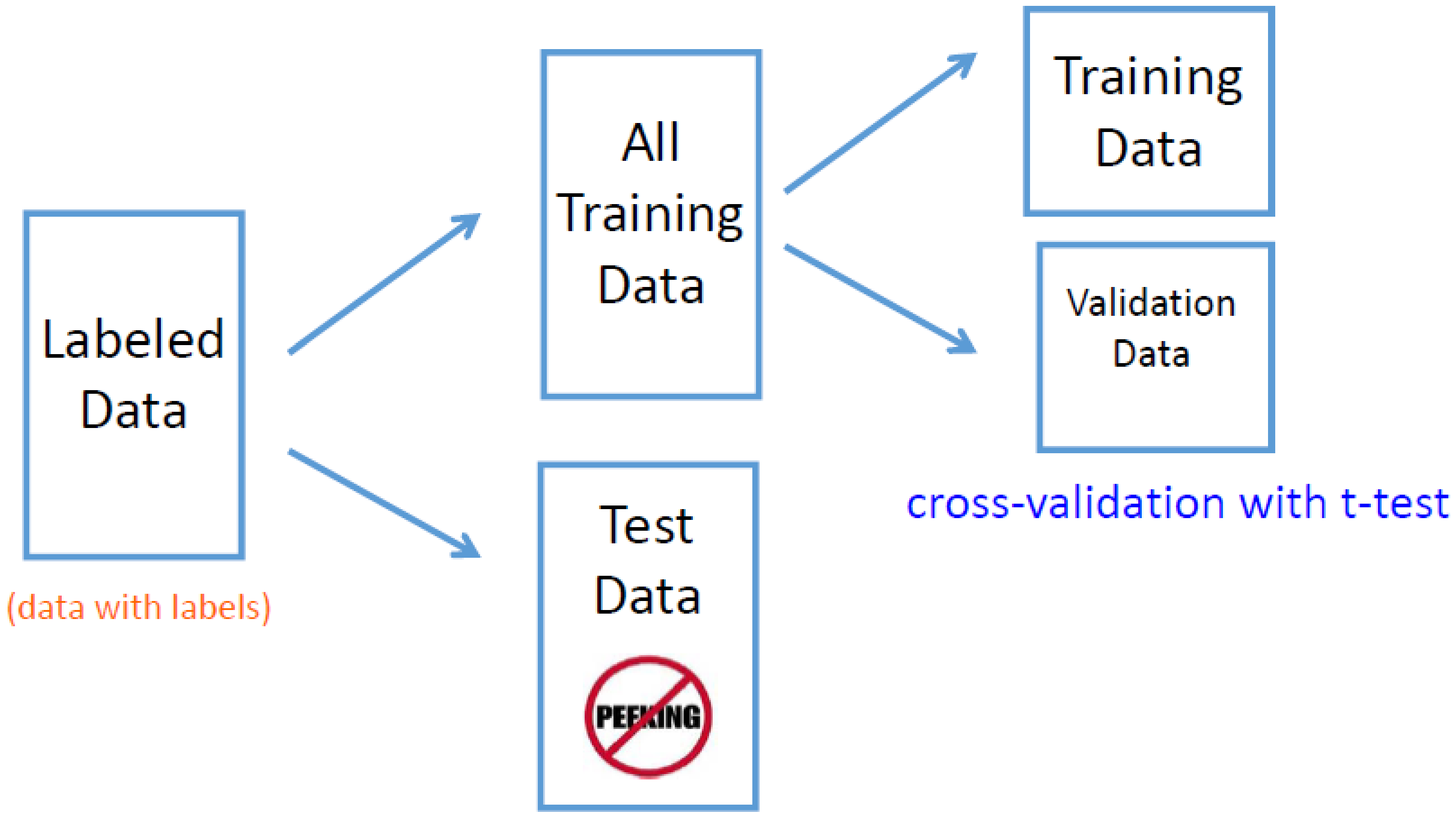
- What values do we use when normalizing testing data?
  - Reuse the same ones from training normalization!

Basic ML Workflow



- Remove noisy features
- Pick “good” features
- Normalize feature values
    - center data
    - scale data (either variance or absolute)
- Normalize example length
- Training
- Testing
- Evaluation

Data

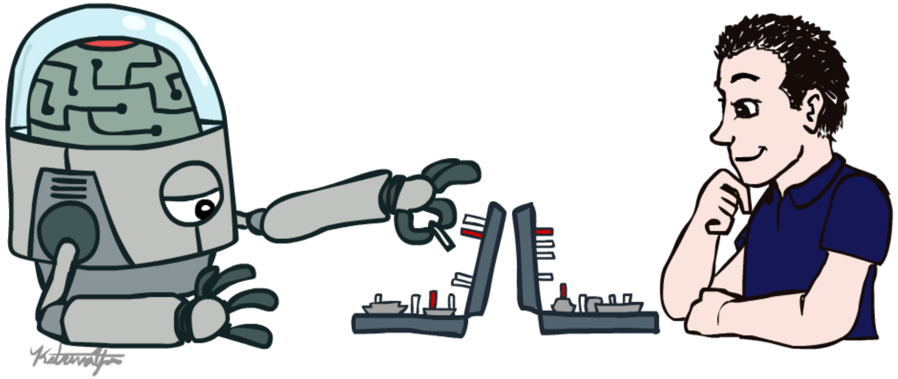

Labeled Data
(data with labels)
All Training Data
Test Data
PEEKING
Training Data
Validation Data
cross-validation with t-test

Experimentation

- **Never look at your test data!**

- During development
  - Compare different models/hyperparameters on validation data
  - use cross-validation to get more consistent results

- For final evaluation, retrain with all training data



Thanks!