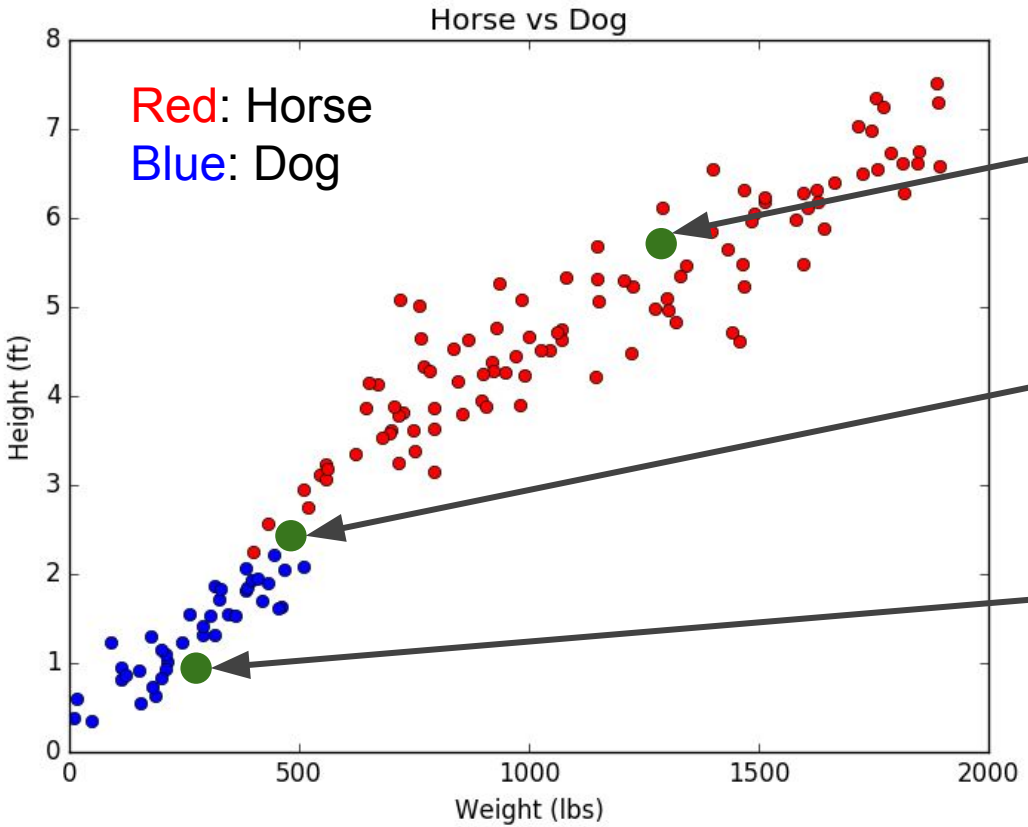
# k-Nearest Neighbors
(kNN)

Ryan Henning

- k-Nearest Neighbors
- The Curse of Dimensionality

# Is it a big dog or a small horse?




New datapoint:
Is it a horse or a dog?

New datapoint:
Is it a horse or a dog?

New datapoint:
Is it a horse or a dog?

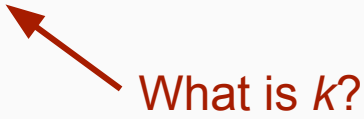
# The k-Nearest Neighbors algorithm:

**Training algorithm:**

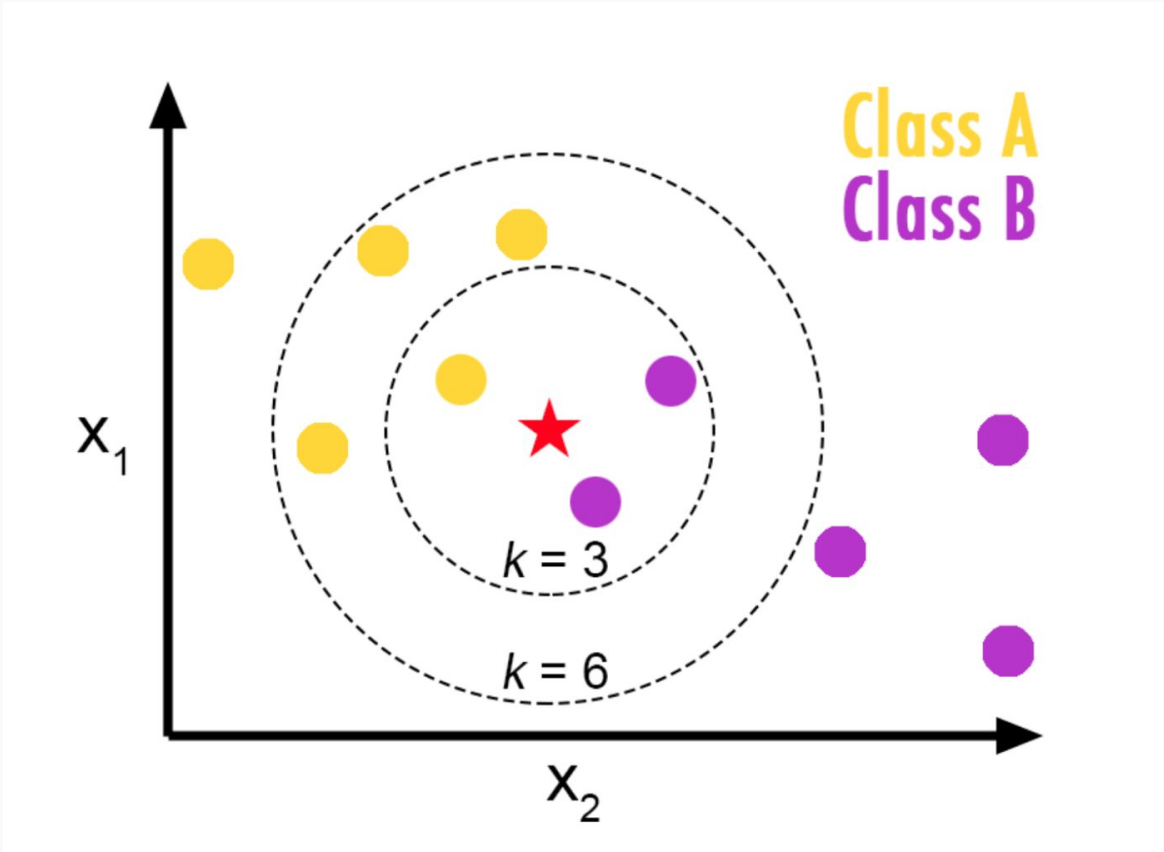
1. Store all the data… that's all.

**Prediction algorithm (predict the class of a new point x'):**

1. Calculate the distance from x' to all points in your dataset.
2. Sort the points in your dataset by increasing distance from x'.
3. Predict the majority label of the *k* closest points.



# kNN: you pick k

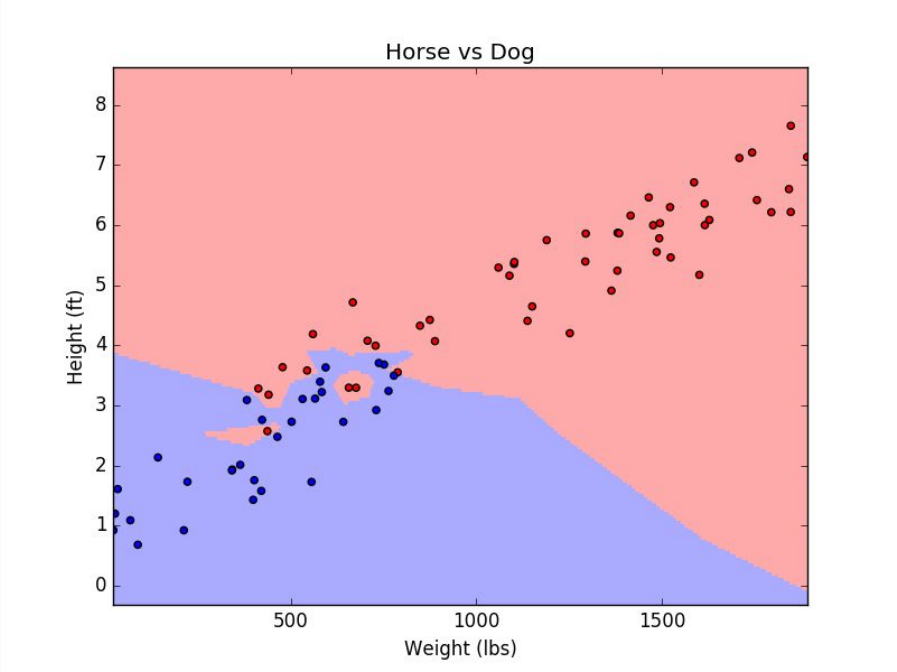



What is the prediction for ★ when k=3?

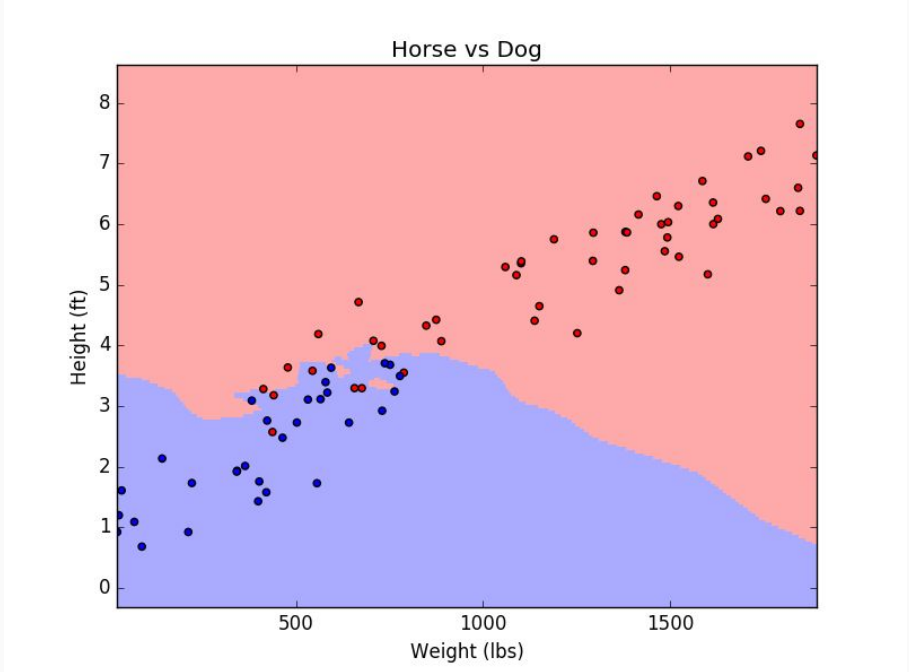
What is the prediction for ★ when k=6?

# The only hyperparameter… *k*… the number of nearest neighbors to consider

**k=1**

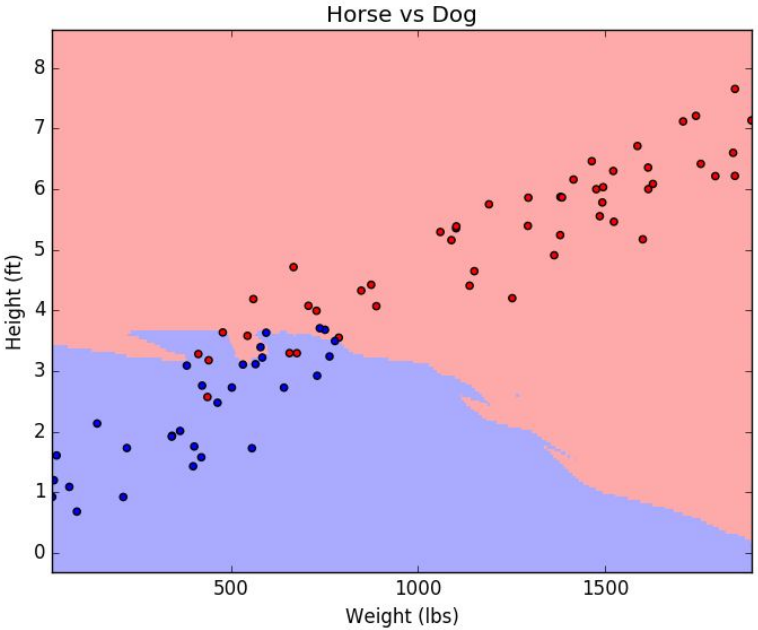



**k=5**

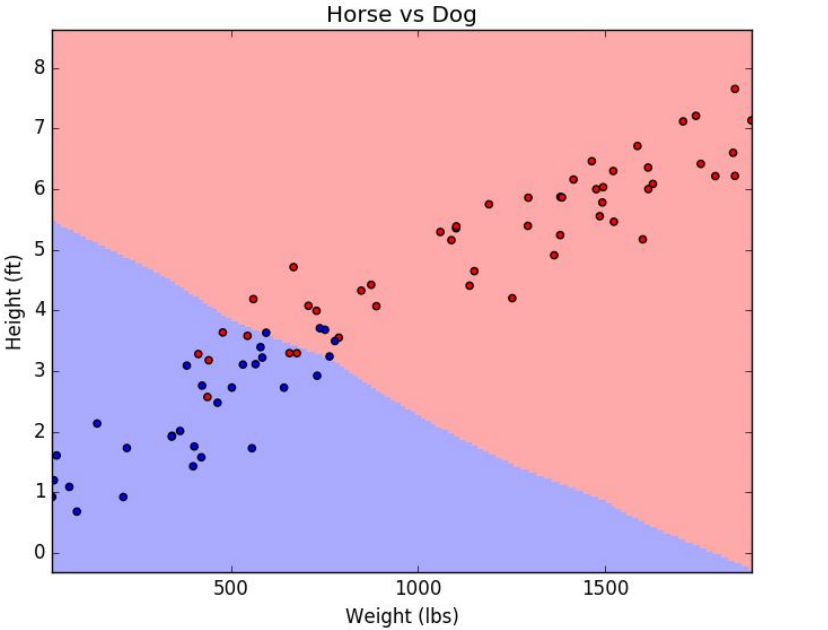


# The only hyperparameter… *k*… the number of nearest neighbors to consider

**k=10**

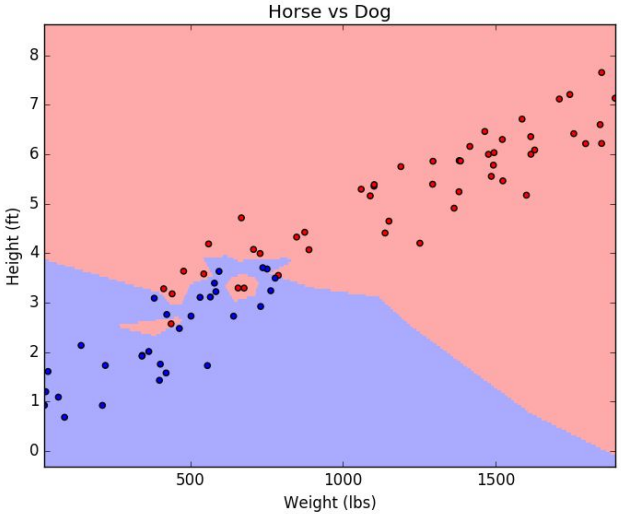



**k=50**



# Which model seems overfit?

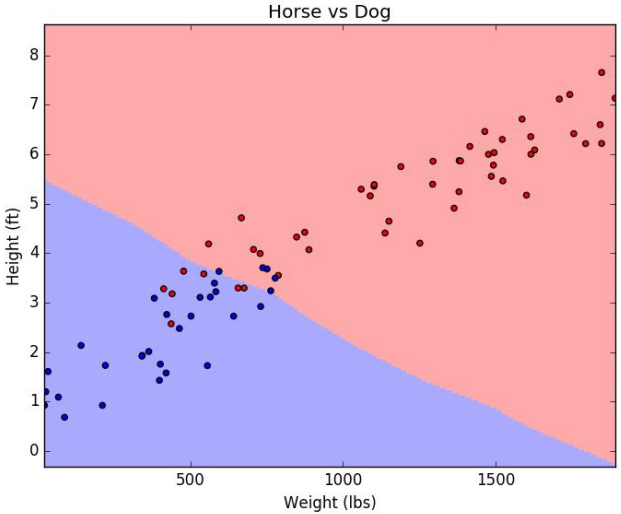
**k=1**




As a general rule, start with:

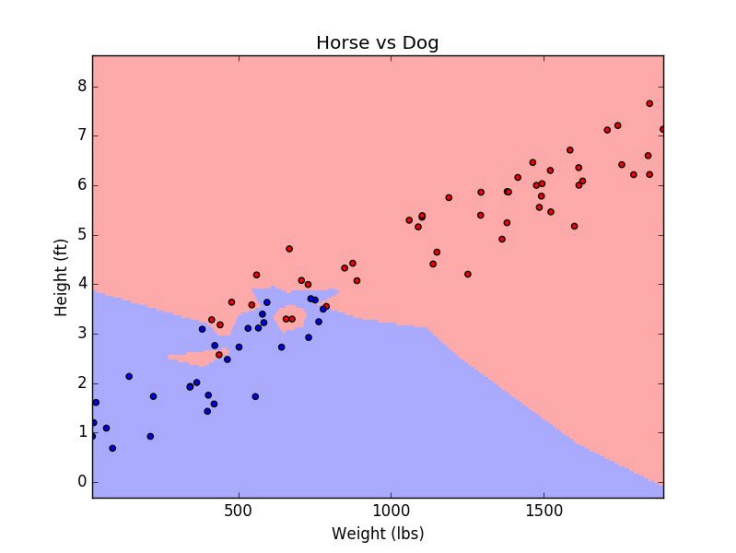
$$k = \sqrt{n}$$

**k=50**

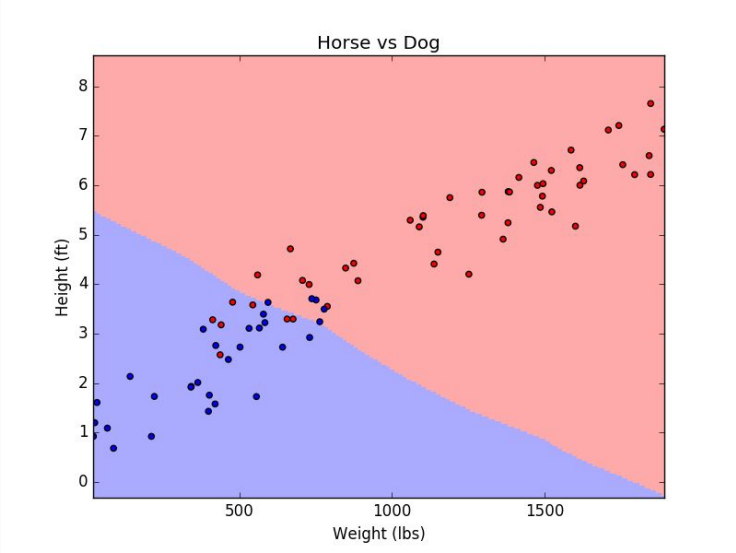


# What happens to model variance when *k* increases?

**k=1**



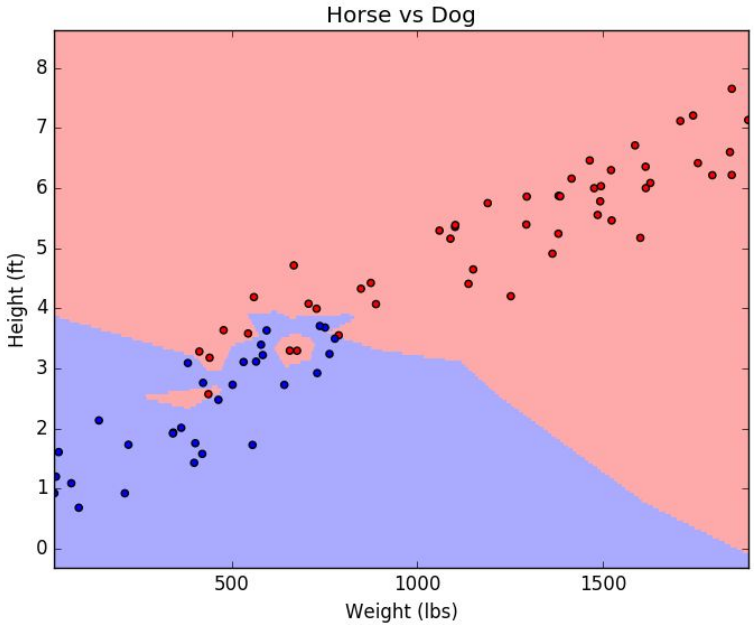

**k=50**



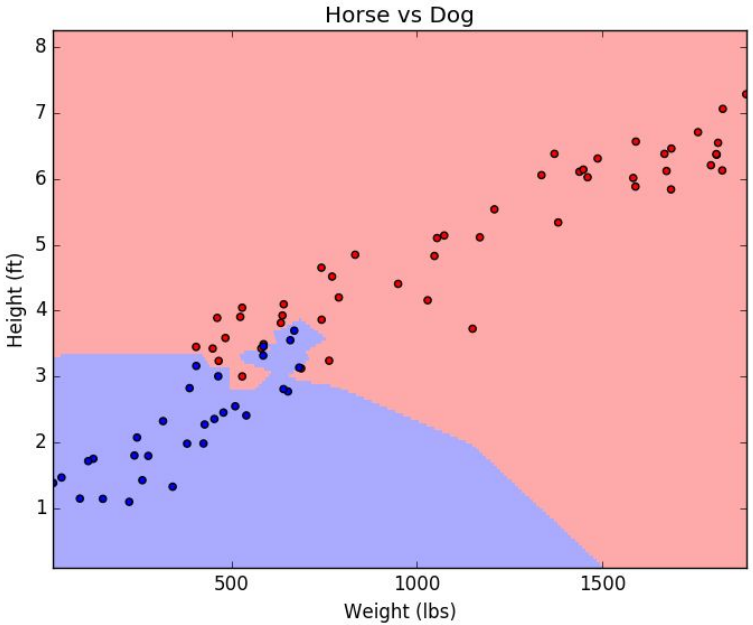
# See the high variance?

Each dataset is randomly generated from the same population.

**k=1**

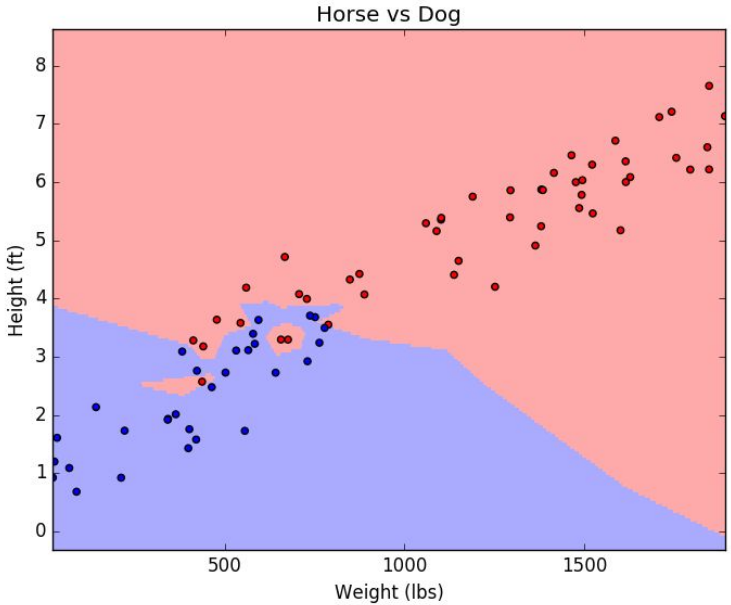



**k=1**

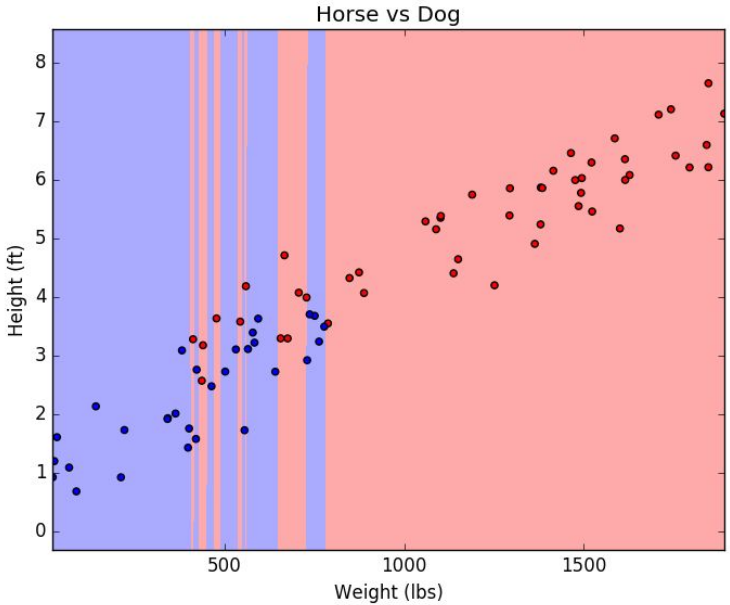


# Be careful with the scale of your features!

## k=1, scaled features




## k=1, original-scale features




Speaking of distance… what distance metric are we using anyway?

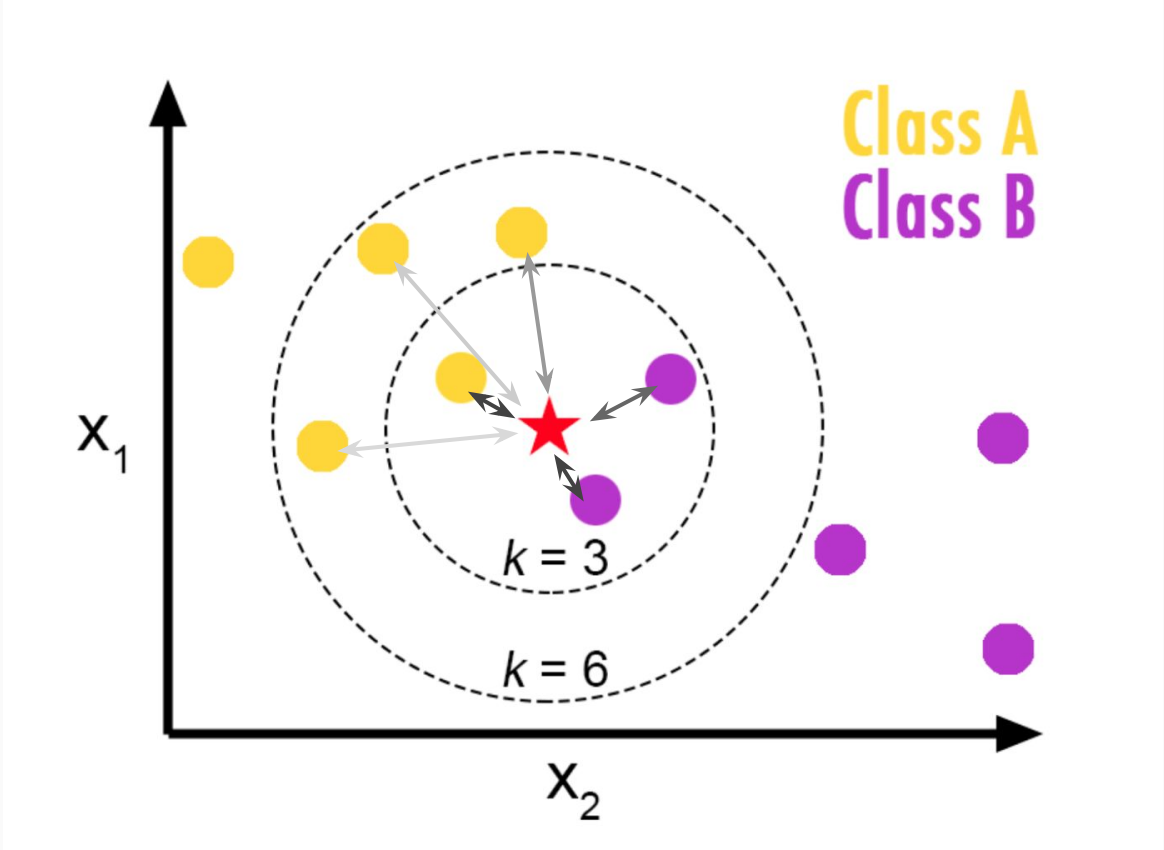
# Distance Metrics

Euclidean Distance (L2): $$\sum_i (a_i - b_i)^2$$

Manhattan Distance (L1): $$\sum_i |a_i - b_i|$$

1 - Cosine Similarity: $$1 - \frac{a \cdot b}{||a|| ||b||}$$

# Point weighting (weighted voting)




Let the *k* nearest points have distances:

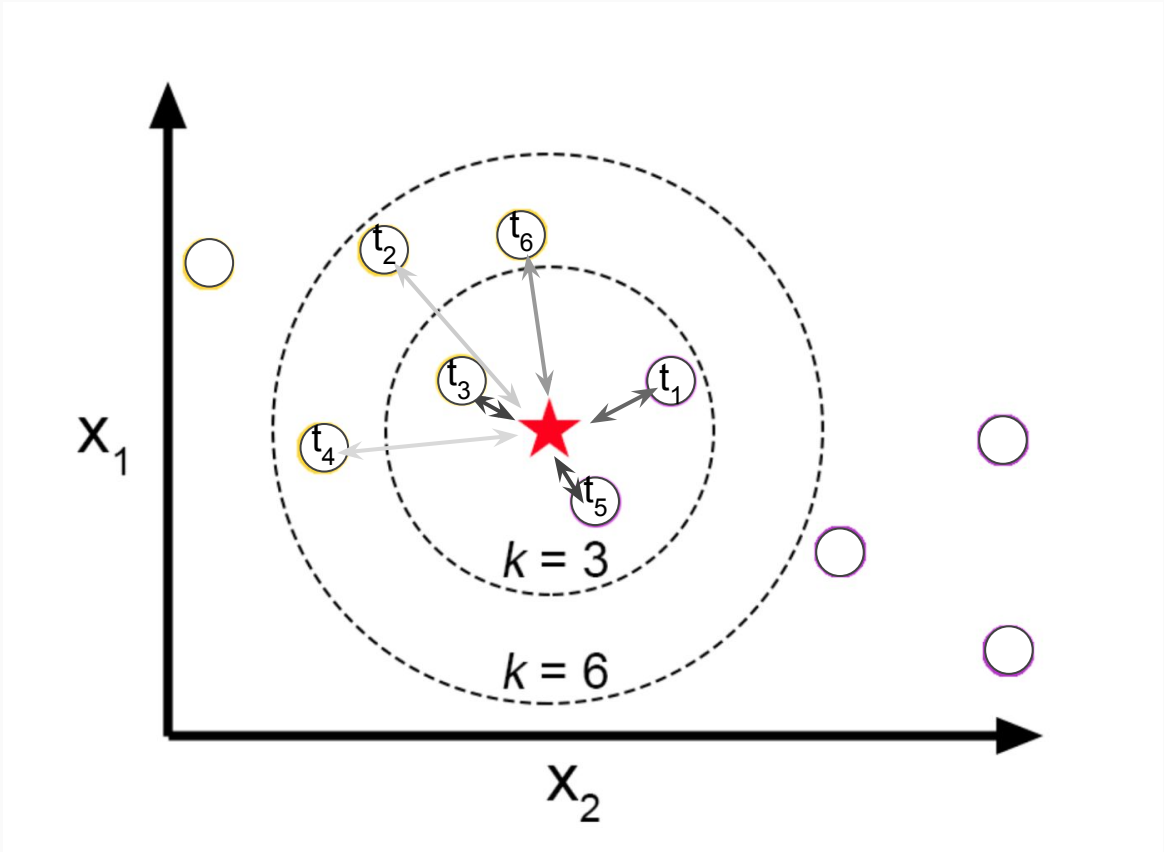
$$d_1, d_2, \ldots, d_k$$

The $i^{th}$ point votes with a weight of:

$$\frac{1}{d_i}$$

small distances are weighted more!

# kNN for Regression




Let the *k* nearest points have distances:

$$d_1, d_2, ..., d_k$$

Let the *k* nearest points have targets:

$$t_1, t_2, ..., t_k$$

Predict the mean value of the k neighbors, or predict a weighted average.

# kNN in high dimensions...

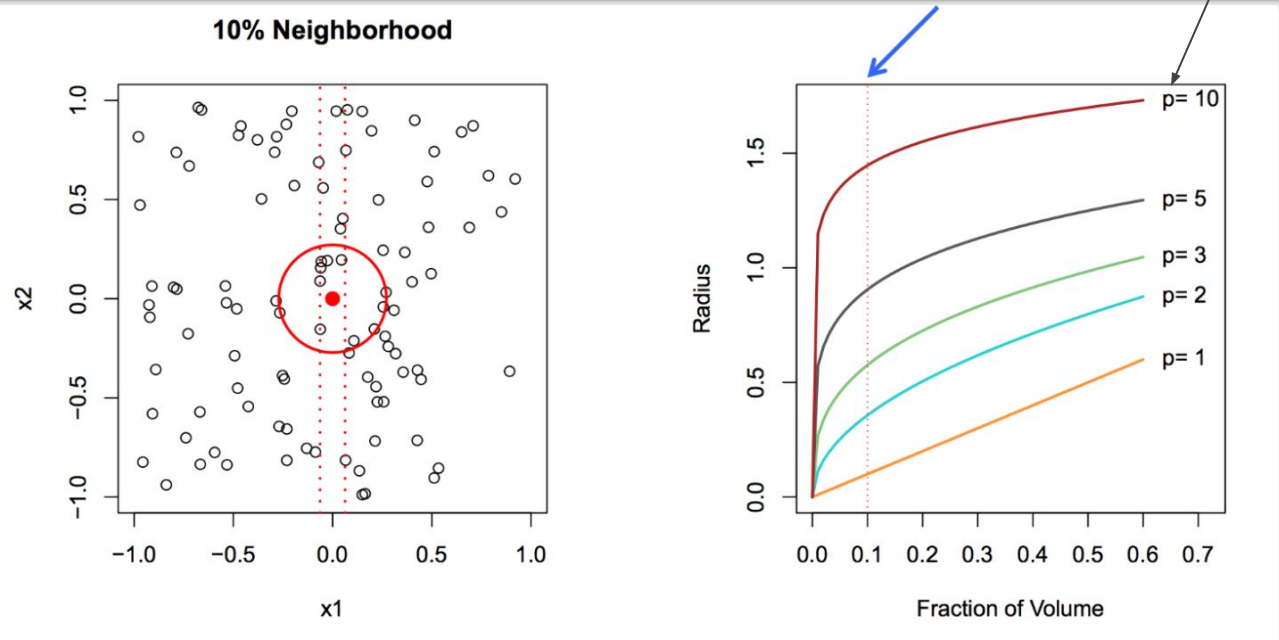
kNN is problematic when used with high dimensional (d) spaces…
... but it works pretty well (in *general*) for d<5

The nearest neighbors can be very **“far” away in high dimensions**…

Say you want to use a neighborhood of 10% (i.e. k = 0.1*n)

Let’s see how this looks as we increase the dimensionality… (next slide)

# The Curse of Dimensionality




When p=1, we are only considering x1. When p=2, we are considering x1 and x2.

Notice the required radius in 2D is much larger than the required radius in 1D.

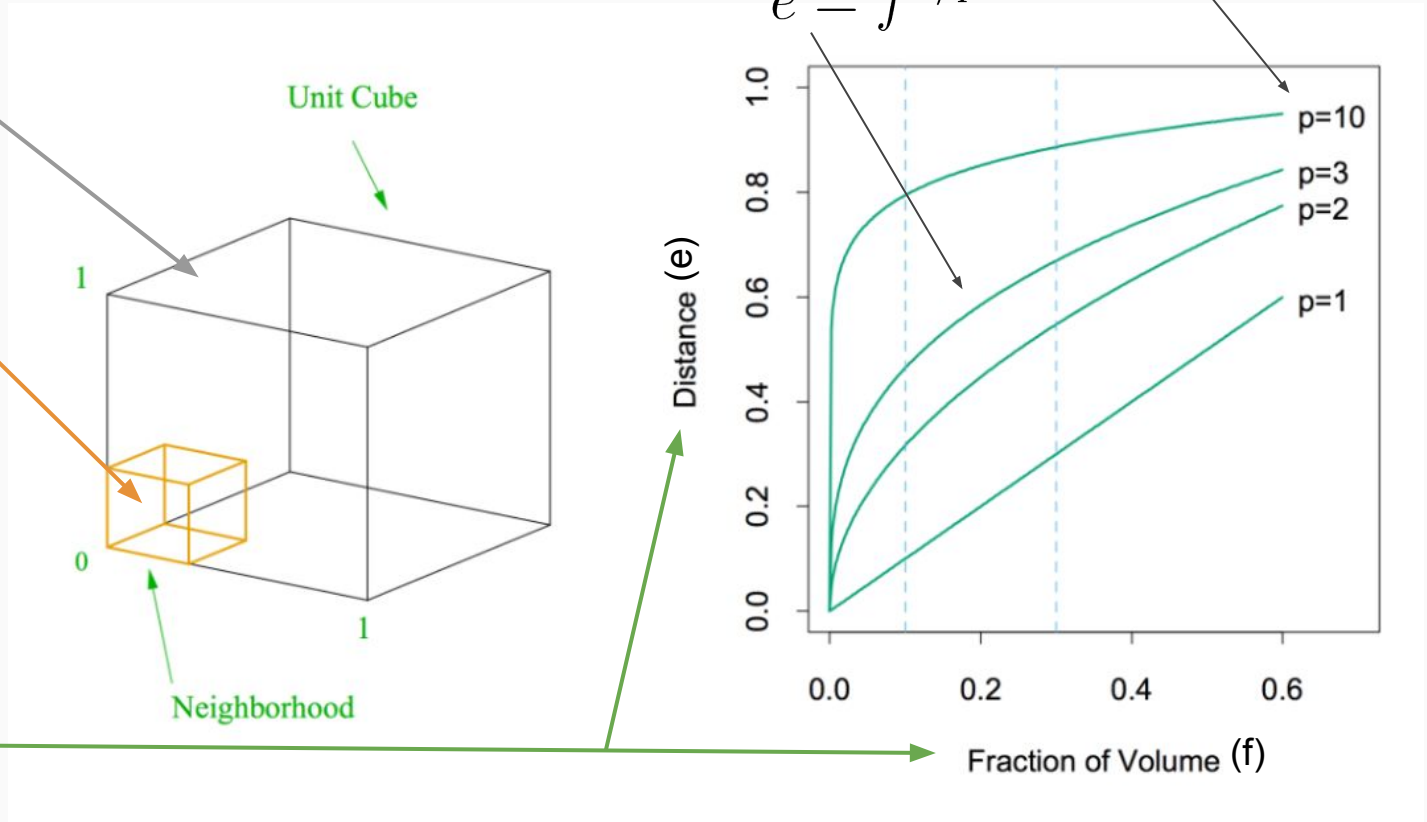
As we increase the dimensionality, we lose the concept of locality.

# The Curse of Dimensionality **(another perspective)**

Say we have a unit (hyper)cube.

We want to create another (hyper)cube inside the outer cube so that we fill X% of the outer cube.

How long must the edges of the inner cube be?



# The Curse of Dimensionality **(another perspective)**

Say you have a dataset with 100 samples, each with only one predictor.

But, one predictor doesn't tell you enough, so you collect a new dataset, and this time you measure 10 predictors for each sample.

How many samples do you need in your new (10 predictor) dataset to achieve the same "sample density" as you originally had (in the one-predictor dataset)?

Just 100^10, that not that many… just

100,000,000,000,000,000,000

# The Curse of Dimensionality **(another perspective)**

Don't freak out...

$$\lim_{d\to\infty} \frac{V_{\text{sphere}}(R,d)}{V_{\text{cube}}(R,d)} = \lim_{d\to\infty} \frac{\frac{\pi^{d/2}R^d}{\Gamma(d/2+1)}}{(2R)^d} = \lim_{d\to\infty} \frac{\pi^{d/2}}{2^d\Gamma(d/2+1)} = 0$$

**Euler's gamma function**…
basically, it's the *factorial* function that can operate on fractional numbers

What does this mean?

Factorial overtakes exponentiation in the limit…
e.g.

$$\lim_{x\to\infty} \frac{c^x}{x!} = 0$$

# The Curse of Dimensionality… takeaways

- kNN (or any method that relies on distance metrics) will suffer in high dimensions.
  - Nearest neighbors are “far” away in high dimensions (even for d=10).
- An 10% neighborhood in a high dimensional unit hypercube requires a hypersphere with large radius.
  - Hyperspheres are weird in high dimensions… I think of them as super-pointy!
- High dimensional data tends to be sparse; it’s easy to overfit sparse data.
  - It takes A LOT OF DATA to make up for increased dimensionality.

# Parametric vs Non-parametric Models

**Parametric models have a fixed number of parameters.**

- Logistic regression is parametric.
- kNN is non-parametric.

Parametric models are more structured. The added structure often combats the curse of dimensionality... as long as the structure is derived from reasonable assumptions.

Alternate perspective: Parametric models are not distance based, so the curse doesn't apply!

# Summary: kNN

Pros:

- super-simple
- training is trivial (store the data)
- works with any number of classes
- easy to add more data
- few hyperparameters:
  - *k*
  - *distance metric*

Cons:

- high prediction cost (especially for large datasets)
- high-dims = bad
  - we'll learn dimensionality reduction methods in two weeks!
- categorical features don't work well…

# Decision Trees

Ryan Henning

- Decision Trees
- Entropy
- Information Gain
- Recursion
- How to build a tree

# Historical log of times I played tennis:

| Temp | Outlook | Humidity | Windy | Played |
|---|---|---|---|---|
| Hot | Sunny | High | False | No |
| Hot | Sunny | High | True | No |
| Hot | Overcast | High | False | Yes |
| Cool | Rain | Normal | False | Yes |
| Cool | Overcast | Normal | True | Yes |
| Mild | Sunny | High | False | No |
| Cool | Sunny | Normal | False | Yes |
| Mild | Rain | Normal | False | Yes |
| Mild | Sunny | Normal | True | Yes |
| Mild | Overcast | High | True | Yes |
| Hot | Overcast | Normal | False | Yes |
| Mild | Rain | High | True | No |
| Cool | Rain | Normal | True | No |
| Mild | Rain | High | False | Yes |

```
def will_play(temp, outlook, humidity,\
              windy):

    if outlook == 'sunny':
        if humidity == 'normal':
            return True
        else: # humidity == 'high'
            return False

    elif outlook == 'overcast':
        return True

    else: # outlook == 'rain'
        if windy == True:
            return False
        else: # windy == False:
            return True
```

# DON'T WRITE CODE LIKE THIS!!!! AHHH!!! #%#%#%@#%!#$^^*%&(%^&*$%^&#$%

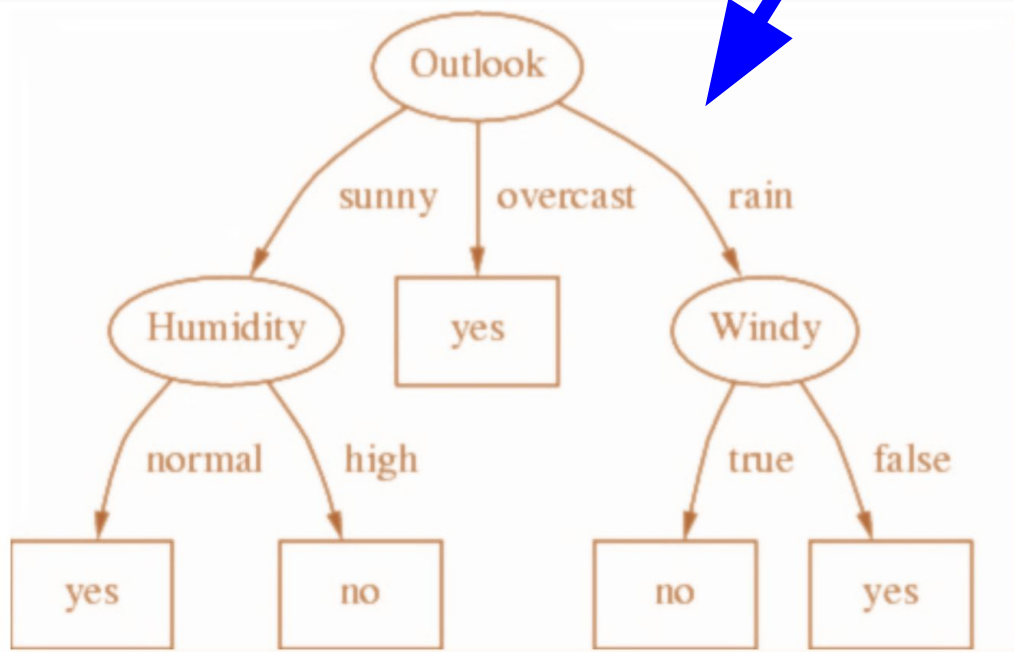
```
def will_play(temp, outlook, humidity,\
              windy):

    if outlook == 'sunny':
        if humidity == 'normal':
            return True
        else: # humidity == 'high'
            return False

    elif outlook == 'overcast':
        return True

    else: # outlook == 'rain'
        if windy == True:
            return False
        else: # windy == False:
            return True
```

Instead, let's write an algorithm to build a **Decision Tree** for us, based on the training data we have.



# Benefits of a decision tree:


## Benefits:

- non-parametric, non-linear
- can be used for classification and for regression
- real and/or categorical features
- easy to interpret
- computationally cheap prediction
- handles missing values and outliers
- can handle irrelevant features

# Drawbacks of Decision Trees


## Drawbacks:

- expensive to train
- greedy algorithm (local maxima)
- easily overfits
- right-angle decision boundaries only

But how can we build one of these from training data?

# Entropy (a measure of information in an event stream)

Shannon Entropy

discrete random variable

information content of X

number of bits needed to encode each X event

$$H(X) = E[I(X)] = E[log_2(\frac{1}{P(X)})]$$

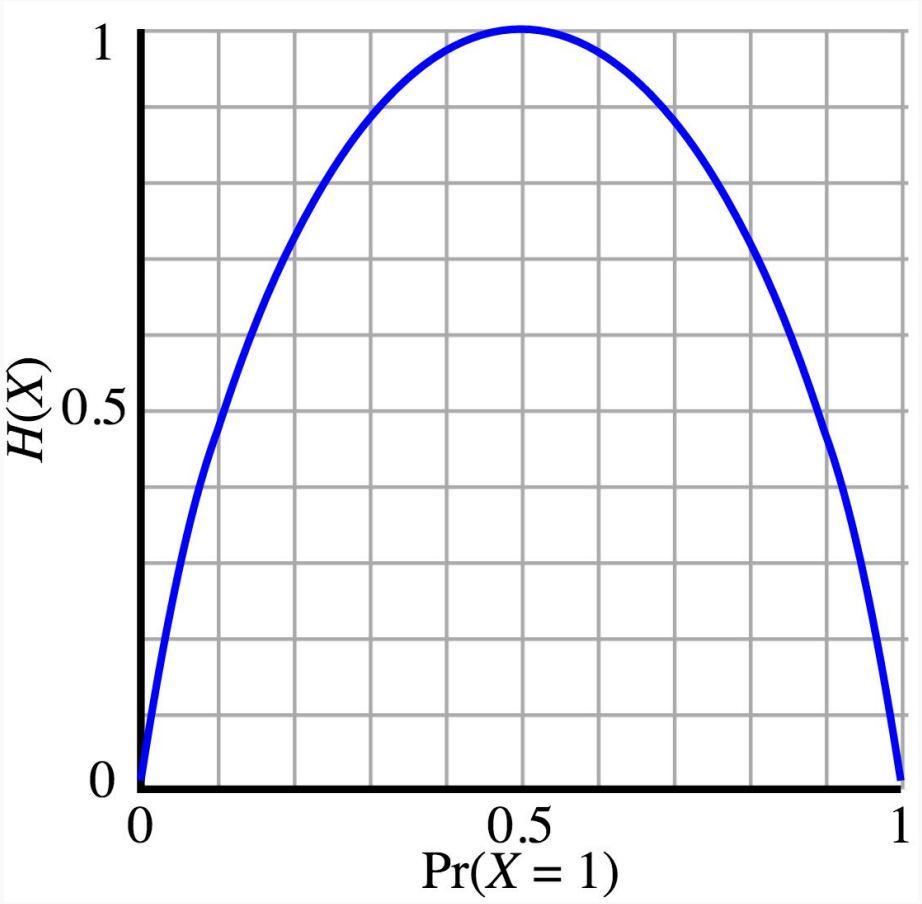
$$= -E[log_2(P(X))]$$

probability of each possible discrete outcome

$$H(X) = -\sum_i p_i log_2(p_i)$$

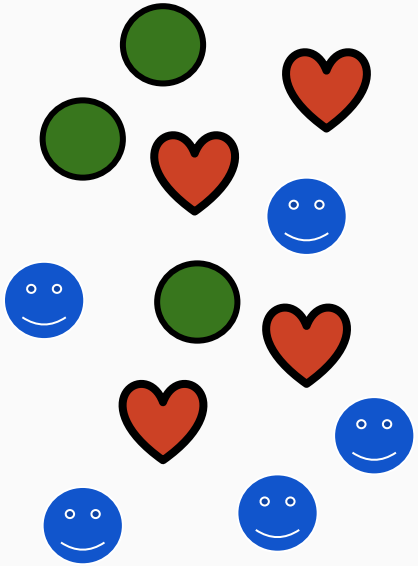
iterate over pmf

# Entropy graph of a Bernoulli random variable X



# Shannon Entropy Diversity Index (aka, the Shannon Index)

We can measure the diversity of a set using Shannon Entropy (H) if we interpret the frequency of elements in the set as probabilities.



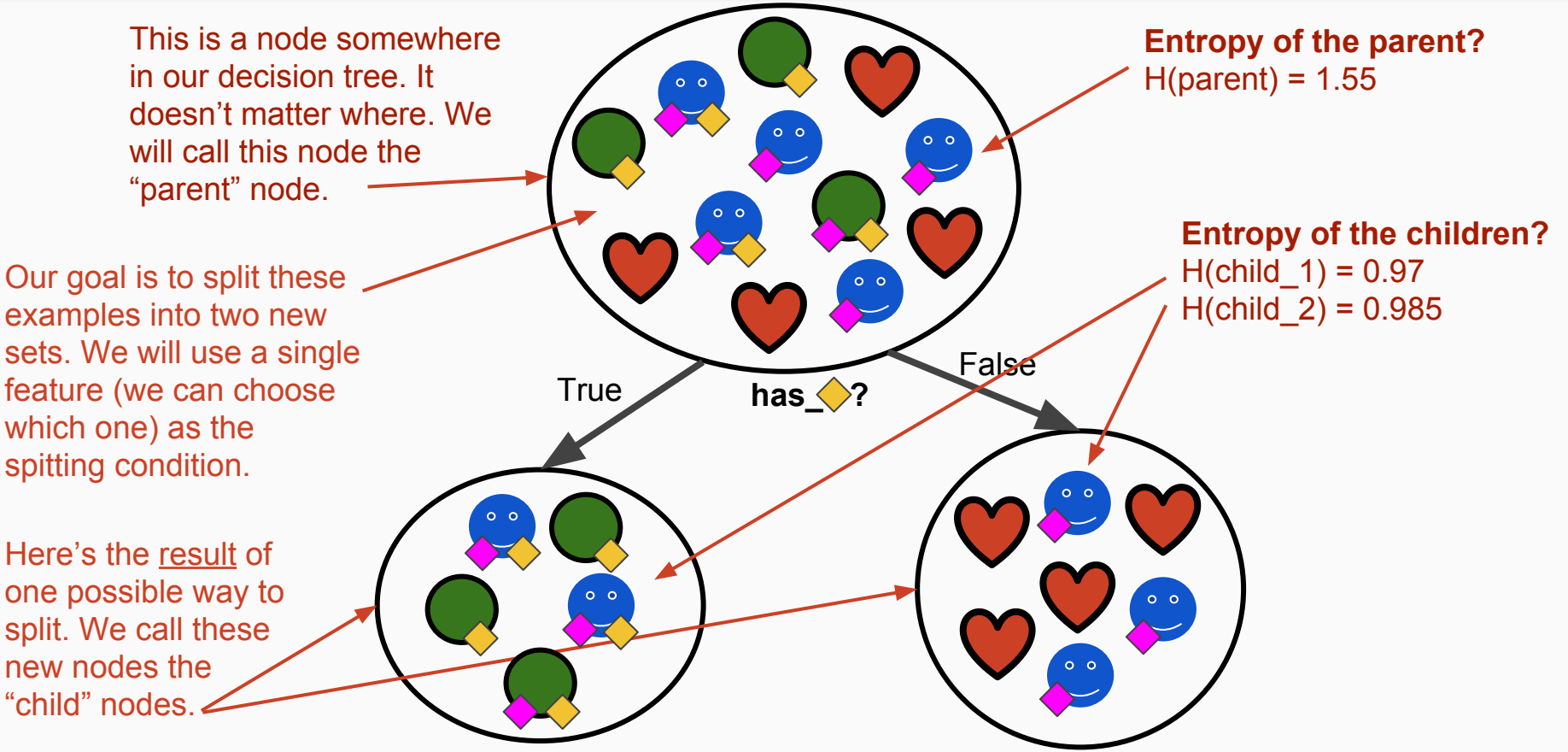
**Estimate:**

P( ) = 3/12 = 0.25
P( ) = 4/12 = 0.33
P( ) = 5/12 = 0.42

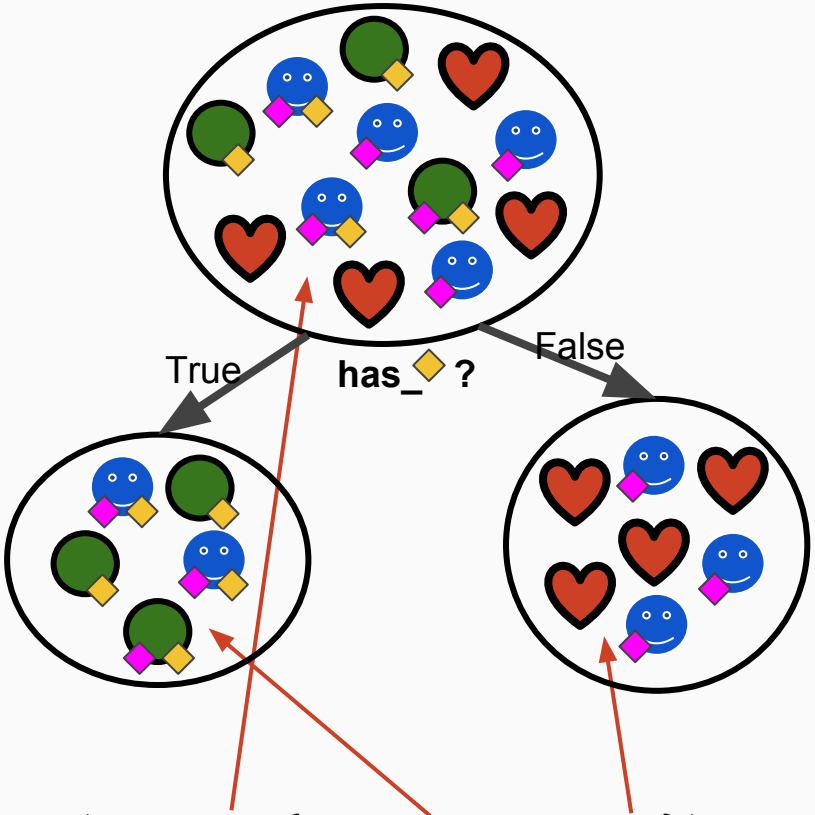
---

H = 1.55

# One level in a decision tree:



# Information Gain (using Shannon Entropy Diversity Index)




Information gain from this split

the set of children

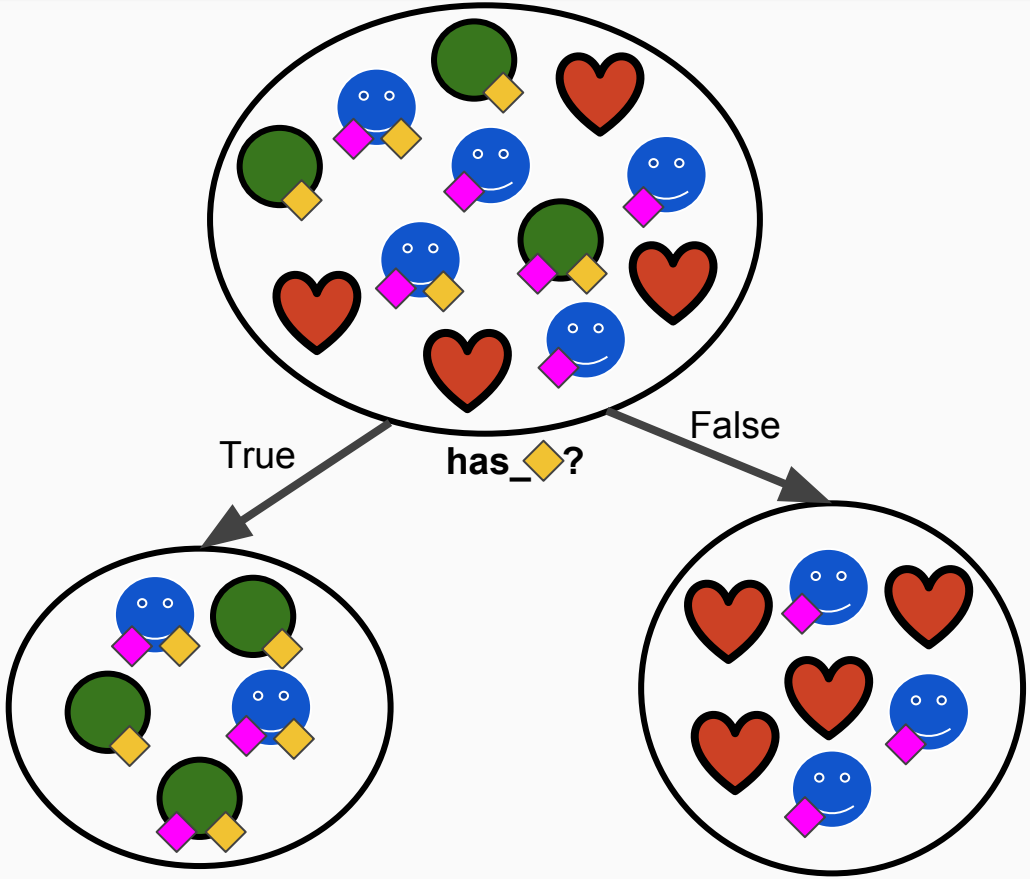
$$\mathrm{IG}(S, C) = H(S) - \sum_{C_i \in C} \frac{|C_i|}{|S|} H(C_i)$$

the parent's set of examples

the set of examples in each child

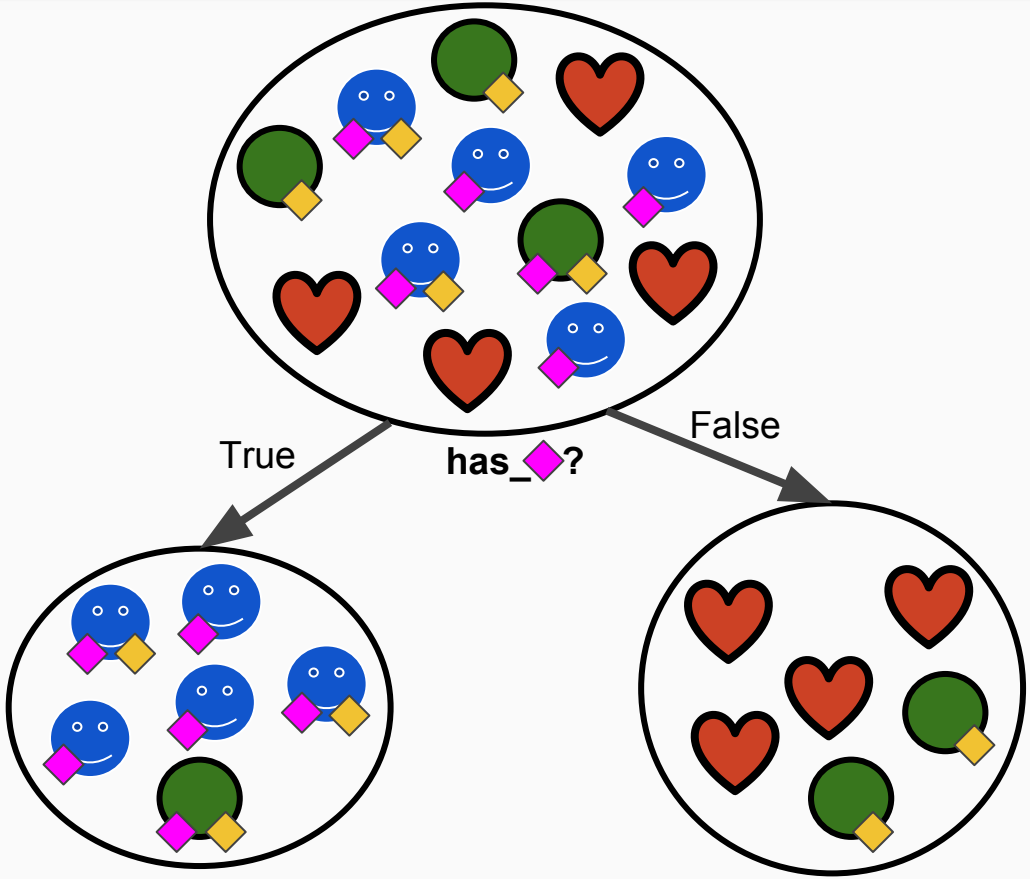
$$\mathrm{IG}(\mathrm{parent}, \{\mathrm{child\_1}, \mathrm{child\_2}\}) = 1.55 - 5/12 * 0.97 - 7/12 * 0.985 = 0.57$$

# Information Gain




Information Gain = 0.57

# Information Gain




Information Gain = 0.765

MORE THAN THE LAST SPLIT. THIS IS GOOD!

# Splitting Algorithm:

**Possible Splits:**

Consider all binary splits based on a single feature:

- if the feature is categorical, split on value or not value.
- if the feature is numeric, split at a threshold: >threshold or <=threshold

**Splitting Algorithm:**

1. Calculate the information gain for all possible splits.

2. Commit to the split that has the highest information gain.

# Recursion

What is this function?

$$f(x) = \prod_{i=1}^{x} i$$

Is this an equivalent function?

$$f(x) = \begin{cases} 1, & \text{if } x \leq 1 \\ xf(x-1), & \text{otherwise} \end{cases}$$

```
def f(x):
    '''
    This function returns x!.
    >>> f(5)
    120
    '''
    if x <= 1:
        return 1
    else:
        return x * f(x-1)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

# How to build a decision tree (pseudocode):

```
function BuildTree:
    If every item in the dataset is in the same class
    or there is no feature left to split the data:
        return a leaf node with the class label
    Else:
        find the best feature and value to split the data
        split the dataset
        create a node
        for each split
            call BuildTree and add the result as a child of the
node
        return node
```

# The Gini Index

A measure of impurity: the probability of a misclassification if a random sample drawn from the set is classified according to the distribution of classes in the set

Scikit-learn doesn't use *Shannon Entropy Diversity* by default. It uses the *Gini Index*:

$$\text{Gini}(S) = 1 - \sum_{i \in S} p_i^2$$

Information gain using the *Gini Index*:

$$\text{IG}(S, C) = \text{Gini}(S) - \sum_{C_i \in C} \frac{|C_i|}{|S|} \text{Gini}(C_i)$$

# Regression Trees

Targets are real values… so…
now we can’t use Information Gain or Gini Index for splitting! What do we do?

Use *variance*! Cool, now we can train.

How do we predict?

Either predict the mean value of the leaf, or do linear regression within the leaf!

# Pruning

Overfitting is likely if you build your tree all the way until every leaf is pure.

Prepruning ideas (prune while you build the tree):

- **leaf size:** stop splitting when #examples gets small enough
- **depth:** stop splitting at a certain depth
- **purity:** stop splitting if enough of the examples are the same class
- **gain threshold:** stop splitting when the information gain becomes too small

Postpruning ideas (prune after you've finished building the tree):

- merge leaves if doing so decreases test-set error
- (see pair.md for details)

# Algorithm Names:

The details of training a decision tree vary… each specific algorithm has a name. Here are a few you'll often see:

- **ID3:** category features only, information gain, multi-way splits, ...
- **C4.5:** continuous and categorical features, information gain, missing data okay, pruning, ...
- **CART:** continuous and categorical features and targets, gini index, binary splits only, …
- ...