https://www.cs.ubc.ca/~fwood/CS340/

Lecture VI

January 20th, 2020
https://www.cs.ubc.ca/~fwood/CS340/lectures/L6.pdf

Laplace Smoothing

Currently we have the following estimate:

$$P(\text{'lactase'} = 1|\text{"spam})= \frac{\#\text{ spam messages with lactose}}{\#\text{ of spam messages}}$$

But there is a problem if there are no spam messages with "lactase". This then means that $P(\text{'lactase'} = 1|\text{"spam}) = 0$, so all spam messages with "lactase" get through automatically.

To fix this we use **Laplace smoothing**:

- Add 1 to the numerator
- add 2 to the denominator

This acts as a 'fake' spam example that has lactase, and a 'fake' spam example that doesn't:

$$\frac{\#\text{ spam messages with lactose} + 1}{\#\text{ of spam messages} + 2}$$

Typically, we do this for all features. It helps against overfitting, by biasing towards the uniform distribution. A common variation is to use a real number β rather than 1. We add βk to the denominator, if the feature has k possible values (so it sums to 1).

$$P(x_{ij} = c|y_i = class) \approx \frac{(\#\text{ of examples with } x_{ij} = c) + \beta}{\#\text{ of examples in class} + \beta k}$$

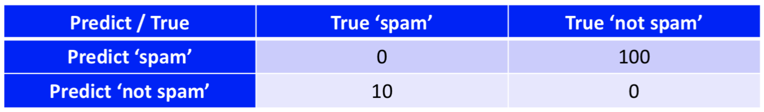
This is a "maximum a posteriori" (MAP) estimate of the probability

Decision Theory

We ask the question of whether we are equally concerned about "spam" and "not spam". For this we consider the cost associated with the different scenarios, namely True positives, true negatives, false positives and false negatives. Costs may vary, since:

- Letting a spam message through (false negative) is not a big deal.
- Filtering a not spam (false positive) message will make users mad.

Consider the following costs:



Predict / True	True 'spam'	True 'not spam'
Predict 'spam'	0	100
Predict 'not spam'	10	0

In this case, instead of taking the most probable model, we use the model $\hat{y}_i$ that minimizes the expected cost:

$$\mathbb{E}[\text{cost}(\hat{y}_i, \tilde{y}_i)]$$

Even if "spam" has a higher probability, predicting "spam" has expected higher cost. **Example:** Consider a test example where we have $P(\tilde{y_i} = \text{"spam"} | \tilde{x_i}) = 0.6$. Then:

$$\begin{aligned}
\mathbb{E}[\text{cost}(\hat{y_i} = \text{"spam"}, \tilde{y_i})] &= P(\tilde{y_i} = \text{"spam"}|\tilde{x_i})\text{cost}(\hat{y_i} = \text{"spam"}, \tilde{y_i}\text{"spam"}) + \\
&\quad P(\tilde{y_i} = \text{"not spam"}|\tilde{x_i})\text{cost}(\hat{y_i} = \text{"spam"}, \tilde{y_i} = \text{"not spam"}) \\
&= (0.6)(0) + (0.4)(100) \\
&= 40 \\
\mathbb{E}[\text{cost}(\hat{y_i} = \text{"not spam"}, \tilde{y_i})] &= (0.6)(10) + (0.4)(0) \\
&= 6
\end{aligned}$$

Even though "spam" is more likely, we should predict "not spam".

Decision Trees vs. Naïve Bayes

Decision Trees

1. Sequence of rules based on 1 feature.
2. Training: 1 pass over data per depth.
3. Greedy splitting as approximation.
4. Testing: just look at features in rules.
5. New data: might need to change tree.
6. Accuracy: good if simple rules based on individual features work ("symptoms").

Naïve Bayes

1. Simultaneously combine all features.
2. Training: 1 pass over data to count.
3. Conditional independence assumption.
4. Testing: look at all features.
5. New data: just update counts.
6. Accuracy: good if features almost independent given label (bag of words).

K-Nearest Neighbors (KNN)

This is an old and simple classifier. To classify an example $\tilde{x_i}$, we:

1. Find the 'k' training examples that are "nearest" to x_i
2. Classify using the most common label of "nearest" training examples.

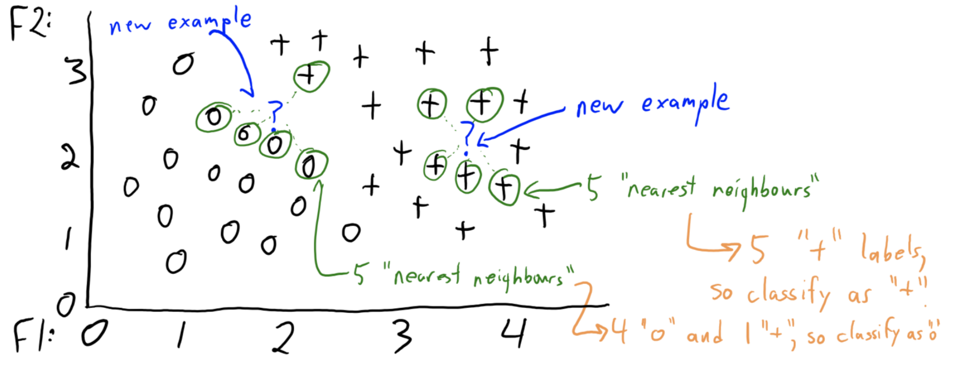


Figure 1: KNN example

Assumption: Examples with similar features, are likely to have similar labels. This seems strong, but basically all good classifiers rely on this assumption:

- If not true there may be nothing to learn and you are in "no free lunch" territory.

- Methods just differ in how you define “similarity”.

The most common distance function is the Euclidean distance:

$$||x_i - \tilde{x_{i'}}|| = \sqrt{\sum_{j=1}^{d}(x_{ij} - \tilde{x_{i'j}})^2}$$

Here, x_i is the feature vector of the training example i, and $\tilde{x_{i'}}$ is the feature vector of test example i'. It costs $O(d)$ to calculate this for pairs of examples.

Effect of k (hyperparameter) in KNN

If k is large, then the model will be very simple, for instance if $k = n$, the we just compute the mode of the labels. The model gets more complicated as k decreases.

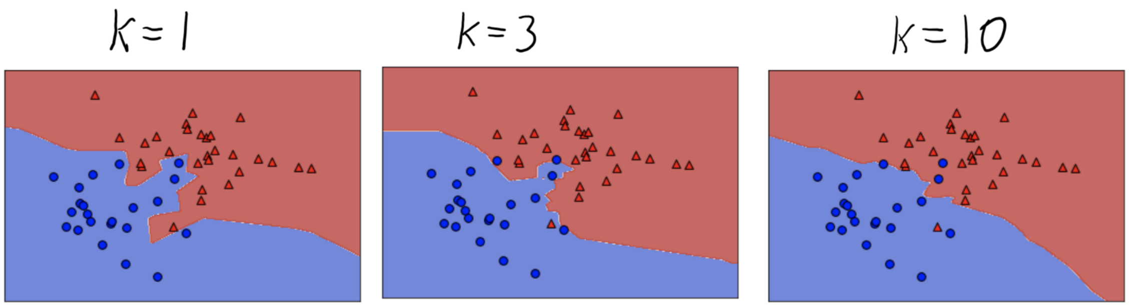


Figure 2: Effect of the value of k on the model

Fundamental trade-off: As k increases, training error increases, but approximation error decreases.

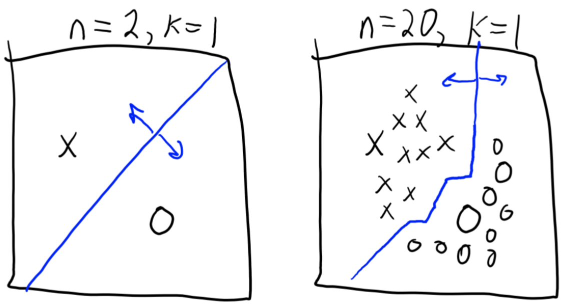


Figure 3: When k is small, the model can be very simple, but gets more complicated as n increases

KNN Implementation

There is no training phase in KNN (lazy learning)

- You just store the training data
- Costs $O(1)$ if you just use a pointer

But Predictions are expensive: $O(nd)$ to classify one example

- Need to do O(d) distance calculation for all ‘n’ training examples.
- So prediction time grows with number of training examples.

But Storage is expensive: needs $O(nd)$ of memory to store X and y.

- So memory grows with number of training examples.
- When storage depends on 'n', we call it a *non-parametric* model

Parametric vs. Non-parametric models

Parametric Models

- Have a fixed number of parameters: trained "model" size is $O(1)$ in terms n.
- You can estimate the fixed parameters more accurately with more data.
- But eventually more data doesn't help: model is too simple

Non-parametric models

- Number of parameters grows with 'n': size of "model" depends on 'n'.
- Model gets more complicated as you get more data.

Parametric models have bounded memory
Non-parametric models have unbounded memory

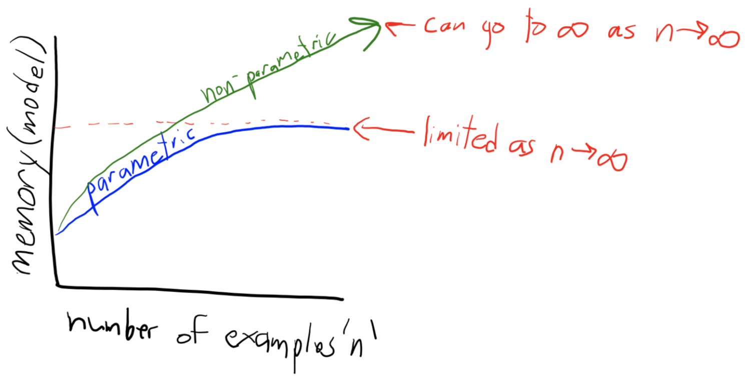


Figure 4: Comparison of memory usage of Parametric vs. Non-parametric models

The curse of Dimensionality

The **curse of dimensionality** is used to describe the problems associated with high-dimensional spaces.

- Volume of space grows exponentially with dimension.
- Need exponentially more points to 'fill' a high-dimensional volume.
  - Nearest" neighbors might be really far even with large n.

Lecture VII

January 22nd, 2020
`https://www.cs.ubc.ca/~fwood/CS340/lectures/L7.pdf`

Norms

Decision Trees vs. Naïve Bayes vs. KNN

Application: Optical Character Recognition

Application: Body-Part Recognition

Ensemble Methods

Ensemble methods are classifiers, that take classifiers as input. They often have higher accuracy than input classifiers.
Remember the fundamental trade off:

1. E_{train}: How small can you make the training error vs.
2. E_{approx}: How well the training error approximate the testing error.

We have that the goal of ensemble methods is that meta classifier:

- Does much better on does much better on one of these, than individual classifiers
- Doesn't do too much worse on the other.

This suggests two types of ensemble methods:

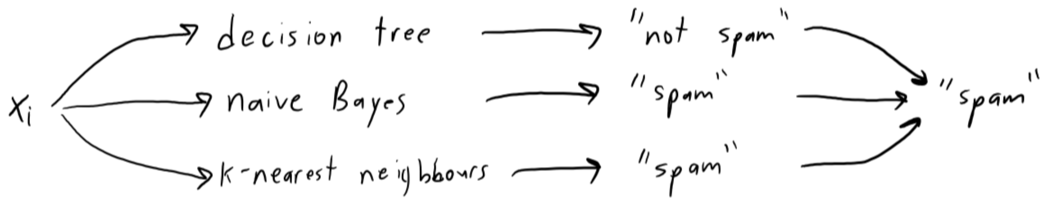
1. Boosting: improves training error of classifiers with high E_{train}
2. Averaging: improves approximation error of classifiers with high E_{approx}

Averaging

The input to averaging is the prediction of a set of models:

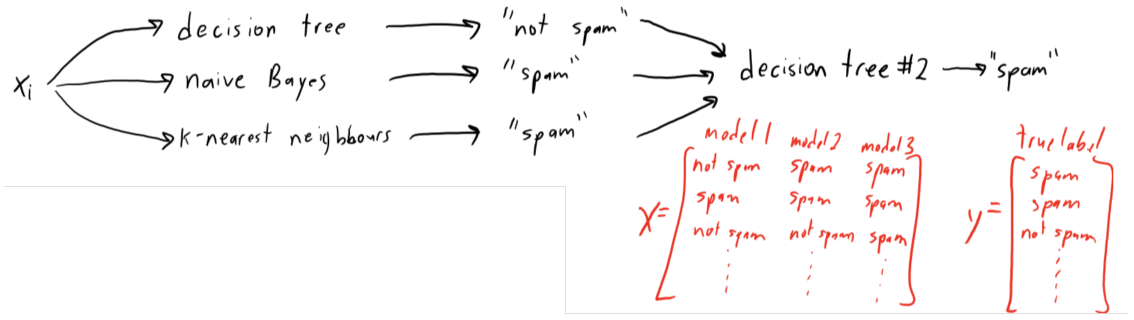
- Decision trees make one prediction.
- Naive Bayes makes another prediction
- KNN makes another

The simple model averaging then takes the mode of the predictions or average probabilities if it is probabilistic.



Digression: Stacking

A common variation of this method is called **stacking**. We fit another classifier that uses the predictions as features.



Why can Averaging work?

Consider 3 binary classifiers, each independently correct with probability 0.80. With simple averaging, ensemble is correct if we have "at least 2 right":

- P(all 3 right) $= 0.83 = 0.512$.
- P(2rights,1wrong) $= 3 \times 0.82(1 - 0.8) = 0.384$.
- P(1 right,2 wrongs) $= 3 \times (1 - 0.8)20.8 = 0.096$.
- P(all 3 wrong) $= (1 - 0.8) \times 3 = 0.008$.
- So ensemble is right with probability 0.896 (which is $0.512 + 0.384$).

Notes:

- For averaging to work, classifiers need to be at least somewhat independent
- You also want the probability of being right to be > 0.5 otherwise it will do much worse.
- We also don't want the probabilities to be too different, otherwise it might be better to use the most accurate one

Intuition If we consider classifiers that over-fit (like deep decision trees), if they all overfit in exactly the same way, then the averaging does nothing.

However if they all make independent errors, then the probability that the average is wrong can be lower than for each classifier, since we pay less attention to specific over-fitting of each classifier.

Random Forests

Random Forests average a set of deep decision trees. It tends to be one of the best "out of the box" classifiers. It's often close to the best performance of any method on the first run, and predictions are very fast.

We have that deep decision trees don't make independent errors, since for each training set the decision tree is the same. therefore we have two key ingredients in Random Forests:

1. Bootstrapping
2. Random trees

Bootstrap sampling

Example We start with a deck of 52 cards, and sample 52 cards with replacement, such that the cards from each of the 52 samples form a new deck of 52 cards (of which some may be duplicates).
The new 52 deck card is called the **"bootstrap sample"**:

- Some cards will be missing, and some cards will be duplicated, so calculations on the bootstrap sample will give different results than original data.
- However, the bootstrap sample roughly maintains trends:
  - Roughly 25%of the cards will be diamonds.
  - Roughly 3/13 of the cards will be "face" cards.
  - There will be roughly four "10" cards.
- A common use is to compute a statistic based on several bootstrap samples. This gives you an idea of how the statistic varies as you vary the data.

Random Forest Ingredient 1: Bootstrap

Bootstrap sample of a list of 'n' examples: a new set of size 'n' chosen independently with replacement:

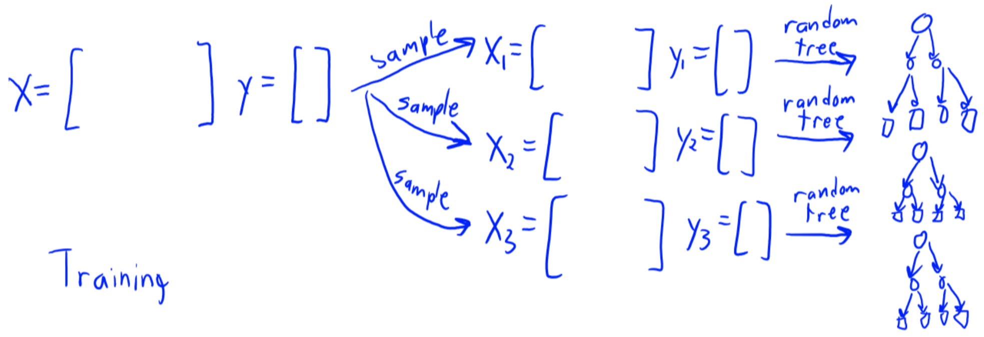
```
for i in 1:n:
  j = rand(1:n)                  # pick a random number from {1, ..., n}
  X_bootstrap[i,:] = X[j,:]    # use the random sample
```

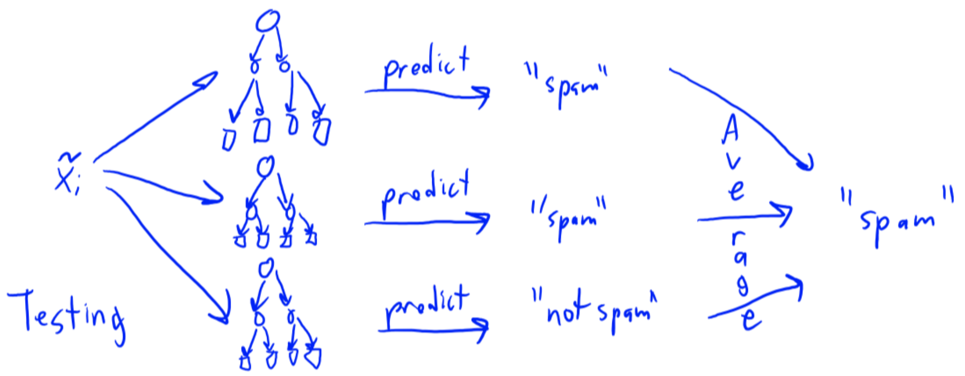
Gives new data set of n examples, with some duplicated and some missing. For large n, approximately 63% of original examples are included.

Bagging: using bootstrap samples for ensemble learning:

- Generate several bootstrap samples of the examples (x_i, y_i)
- Fit a classifier to each bootstrap sample.
- At test time, average the predictions.

Therefore **Random Forests** are an ensemble method that averages the results of fitting deep random trees to bootstrap samples of data. The randomization **encourages errors of different trees to be independent**.

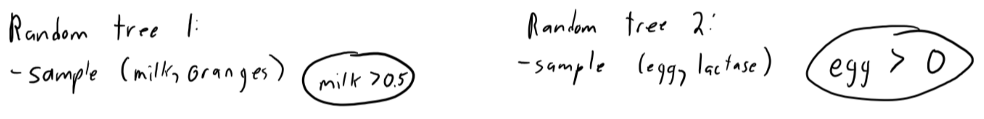


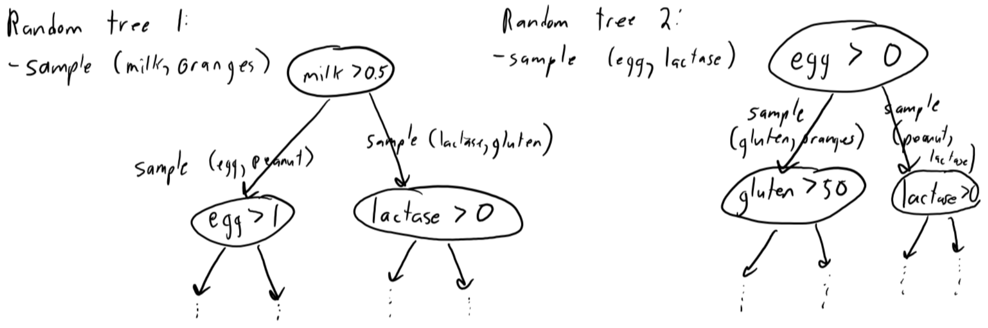


Random Forest Ingredient 2: Random Trees

For each split in a random tree:

- randomly sample a small number of possible features (typically $\sqrt{d}$).
- Only consider these random features when searching for the optimal rule





- Splits will tend to use different features in different trees. they will still overfit, but hopefully errors will be more independent.
- So the average tends to have a much lower test error.
- Empirically,random forests are one of the “best” classifiers.

Lecture VIII

January 24th, 2020
`https://www.cs.ubc.ca/~fwood/CS340/lectures/L8.pdf`

Unsupervised learning

In **Supervised learning**, we have:

- Features x_i and class labels y_i
- We write a program that produces y_i from x_i.

In **Unsupervised learning**:

- We **only have x_i values**, but no specific target labels
- We want to do something with them

Some *supervised learning tasks are*:

- Outlier detection: Is it a 'normal' x_i?
- Similarity search: which examples look like this x_i?
- Association rule: Which x^j occur together?
- Latent-factors: What 'parts' are the x_i made from?
- Data visualization: What does the high-dimensional X look like?
- Ranking: Which are the most important x_i?
- Clustering: What types of x_i are there?

Clustering

Input: set of examples described by features x_i

Output: an assignment of examples to 'groups'

Note: Unlike classification, we are not given the groups. This means that the Algorithm must discover the groups

Example

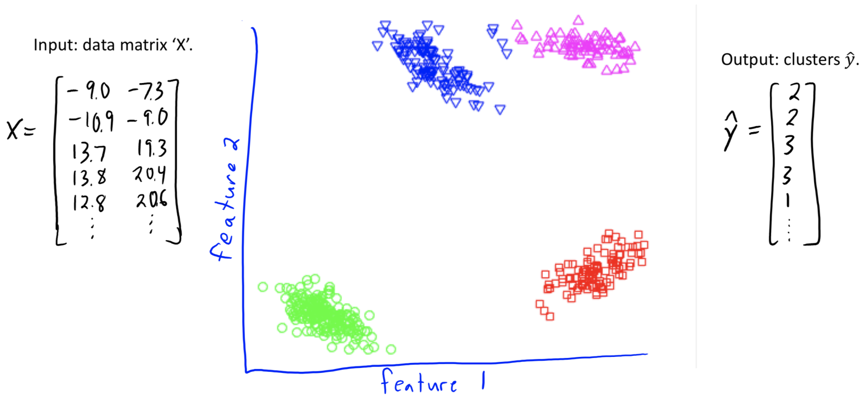

Input: data matrix 'X'.
$X = \begin{bmatrix} -9.0 & -7.3 \\ -10.9 & -9.0 \\ 13.7 & 19.3 \\ 13.8 & 20.4 \\ 12.8 & 20.6 \\ \vdots & \vdots \end{bmatrix}$
feature 2
feature 1
Output: clusters $\hat{y}$.
$\hat{y} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 3 \\ 1 \\ \vdots \end{bmatrix}$

Figure 5: Example of clustering

Data clustering

The general goal of clustering algorithms is that examples in the **same group should be 'similar'**, and examples in **different groups should be 'different'.**

But the 'best' clustering is hard to define, since we don't have a test error. This means that generally there is no best method in unsupervised learning: so there are lots of methods and we'll focus on important/representative ones.

So why cluster?

- We could want to know what the different groups are.
- We could want to find the group for a new example x_i.
- We could want to find examples related to a new example x_i.
- We could want a 'prototype' example for each group.

K-means

The most popular clustering technique is called K-means. As **Input** we have:

- The number of clusters 'k' (hyper-parameter)
- Initial guess of the center (the mean) of each cluster

The **Algorith works as follows**:

- Assign x_i to its closest mean
- Update the means based on the assignment
- Repeat until convergence

Example

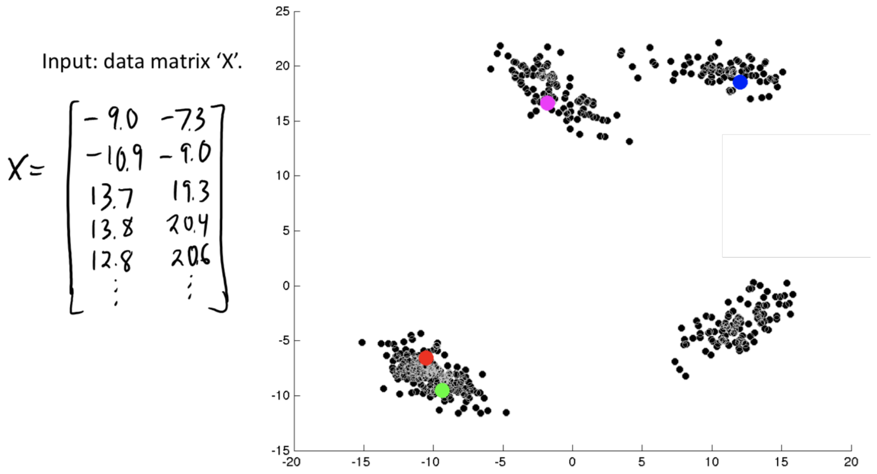


Figure 6: Start with k initial 'means' (usually, random data points)

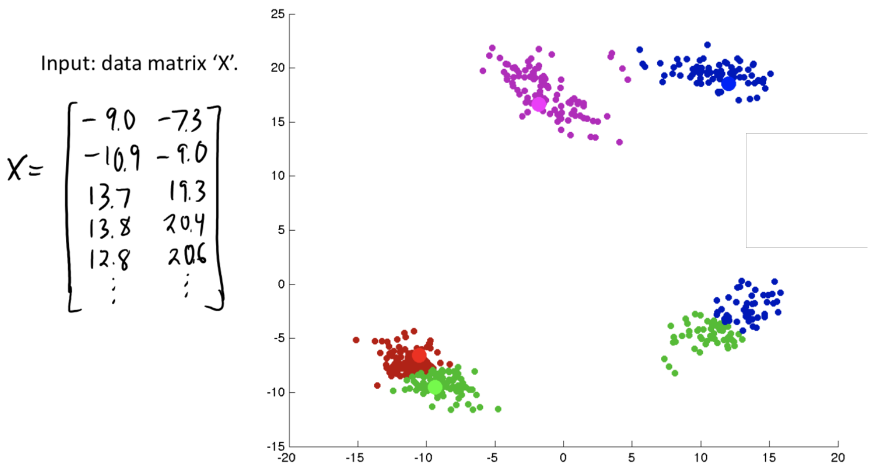


Figure 7: Assign each example to the closest mean.

We then update the mean of each group, until no data points change groups

K-Means Issues

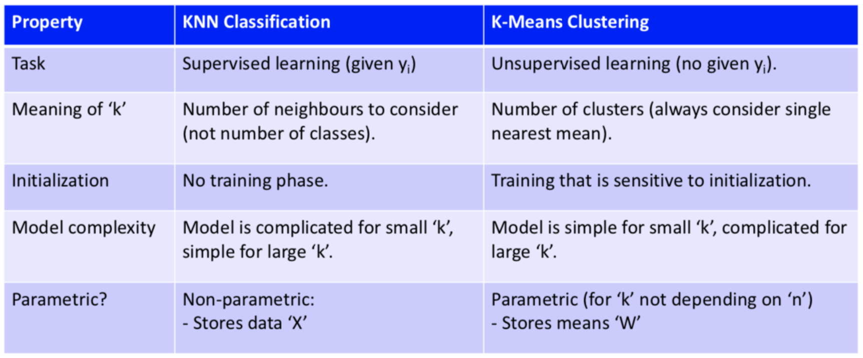
K-means is guaranteed to converge when we use euclidean distance. Given a new test example, we assign it to the nearest mean to cluster it. However:

- K-means assumes that you know the number of clusters k: there are lots of heuristics to pick k, but none are satisfying.
- Each example is assigned to one, and only one cluster. There is no possibility of overlapping clusters or leaving examples unassigned.
- It may converge to a sub-optimal solution.

Since k-means may depend on how the problem was initialized, then it is good practice to start the problem with different initializations, and find the best fit. This is called random restarts, where we choose several random initialization points, and choose the best one.

KNN vs. K-means

They are not to be confused!!!



Property	KNN Classification	K-Means Clustering
Task	Supervised learning (given y_i)	Unsupervised learning (no given y_i).
Meaning of 'k'	Number of neighbours to consider (not number of classes).	Number of clusters (always consider single nearest mean).
Initialization	No training phase.	Training that is sensitive to initialization.
Model complexity	Model is complicated for small 'k', simple for large 'k'.	Model is simple for small 'k', complicated for large 'k'.
Parametric?	Non-parametric: - Stores data 'X'	Parametric (for 'k' not depending on 'n') - Stores means 'W'

Figure 8: Table showing the difference between KNN and K-means

What is K-Means doing?

We can interpret K-means as minimizing an objective function, which is the sum of squared distances from each example x_i to its center $w_{\hat{y}_i}$:

$$\text{min.} \quad f(w_1, w_2, ..., w_k, \hat{y_1}, \hat{y_2}, ..., \hat{y_n}) = \sum_{i=1}^{n} \|w_{\hat{y_i}} - x_i\|^2$$

In the above function, we have that $w_{\hat{y_i}}$ is the cluster of example i, therefore we have that $\hat{y}_i \in \{1, ..., k\}$. Consequently the two k-means steps are:

- Minimize f in terms of the $\hat{y}_i$ (i.e.: update the cluster assignments).
- Minimize f in terms of the w_c (i.e.: update the means)

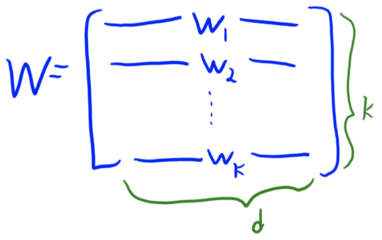


Figure 9: Table showing the difference between KNN and K-means

The termination of the algorithm follows because:

- Each step does not increase the objective.
- There are a finite number of assignments to k clusters.