kMeans HAC Clustering-

Kmeans algorithm

Given a dataset, **k-Means splits it into k disjoint groups.**

- Setting k is largely heuristic as larger k produces lower SSE
  - Unsupervised learning has no validation set because it has no labels
- Guaranteed to converge in finite steps to a local minima
  - Often in a few iterations in practice
- O(mknd) computational complexity makes running k-Means tractable

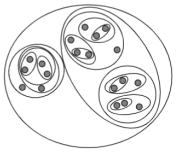
Properties of k-means:

- Not particularly resistant to outliers
- Very sensitive to initialization of centroids – need to run multiple times
- Only seems to work on spherical clusters with similar sizes

HAC clustering algorithm

Given a dataset $X = \{\boldsymbol{x}_i\}_{i=1}^{n}$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ and some distance function $d(x_i, x_j)$ which measures the distance between two points:

1. Initialize every datapoint as its own cluster
2. Until there is only one cluster remaining:
   a) Merge the two closest clusters



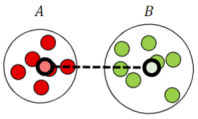
Notice it doesn't output a specific number of clusters. Can save intermediate clusterings from k=n to k=1.

Question: We have a measure of distance between points, how do we use it to measure "closeness" of clusters?

Centroid

- The distance between the cluster means

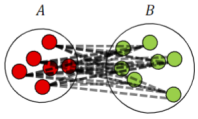
$$d(A, B) = d\left(\overline{x_a}, \overline{x_b}\right)$$



Average-link

- Average distance between all cross-cluster pairs

$$d(A, B) = \frac{1}{|A||B|} \sum_{x_a \in A} \sum_{x_b \in B} d(x_a, x_b)$$



- **Most robust and most commonly used**

HAC is a convenient tool that often provides interesting views of a dataset

Primarily HAC can be viewed as an intuitively appealing clustering procedure for data analysis/exploration

We can create clusterings of different granularity by stopping at different levels of the dendrogram

HAC often used together with visualization of the dendrogram to decide how many clusters exist in the data

Different linkage methods (single, complete and average) often lead to different solutions

Overfitting in decision trees-

How to avoid overfitting

Option 1: Add more hyperpameters to control tree size:

- Limit depth / limit number of nodes / only split if at least K datapoints present

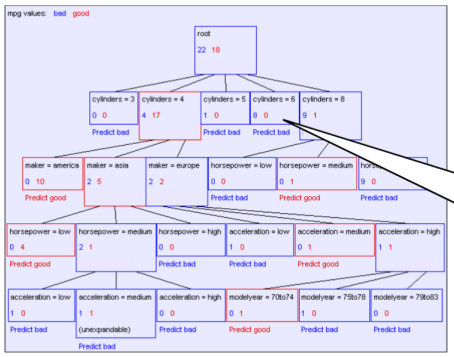
Option 2: Early stopping based on validation performance

- Monitor the validation accuracy and stop splitting a branch when performance saturates.
- Can be tricky for the same reasons that our proposed Base Case 3 can be.

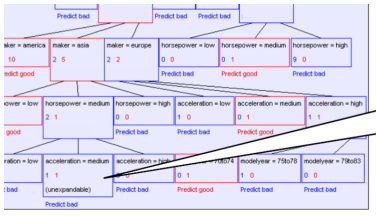
Option 3: Post Pruning

- Grow full tree on the training set, then consider the impact of removing each node on validation performance.
- Greedily prune the node that most improves validation set performance.

When do we stop? **Base Case 1**



Don't split a node if all matching records have the same output value



Don't split if none of the attributes can divide the examples

BasicGreedyAlgorithm(dataset S):

If S all have same label or all same features:

return Leaf(majorityLabel(S))

Based on S, choose "best" test t to split the data:

Evaluate the information gain of possible splits and pick the largest

Split S into subset $S_1, \ldots, S_k$ for each unique outcome of t applied to S

For i in {1,…,k}:

Create child node c_i = BasicGreedyAlgorithm(S_i)

Dimensionality Reduction with PCA-

How to implement:

Say you have data that is 100 dimensional and you want it down to 3.

```
# X is an n x d data matrix
X = X - np.mean(X,axis=0)
cov = X.T @ X / n
eigvals, eigvecs = np.linalg.eig(cov)
```

To perform PCA on your data matrix X (n x 100):

1. Compute the mean vector of your data (100 dim vector) and subtract it from X to center your date
2. Compute the covariance matrix by computing $\frac{1}{n}X^TX$
3. Call a package to compute the Eigenvalues/Eigenvectors of the covariance.
4. Sort both in descending order by the Eigenvalues and return Eigenvectors corresponding to the top k. Usually as a (d x k) matrix W with each column being one Eigenvector.

- Begin by describing the steps of PCA in detail, emphasizing the importance of standardization, covariance matrix computation, and eigenvalue/eigenvector analysis.
- Explain how to choose the number of principal components, considering explained variance and potential methods for determining the optimal number.
- Discuss the benefits of PCA, such as reduced dimensionality and improved computational efficiency, and address limitations, such as the assumption of linearity.
- Provide a hypothetical scenario where PCA might be beneficial, for example, in visualizing high-dimensional data or improving the efficiency of a machine learning algorithm.
- Conclude by summarizing the key points related to PCA's application, benefits, and limitations.

Eigenvectors/values explained

Eigenvectors represent directions in feature space.

Eigenvalues represent the importance (magnitude of variance) of those directions.

Principal components are the eigenvectors ordered by their associated eigenvalues.

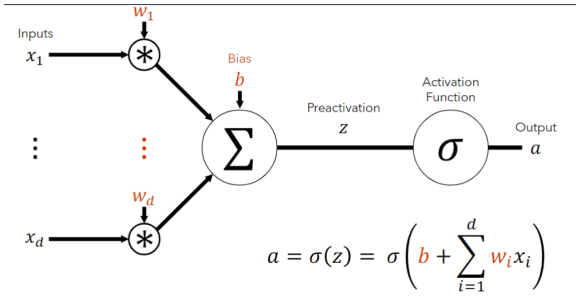
Goal of PCA:

1. **Projection onto $\mathbf{w}$:**
   - Each data point $\mathbf{x}_i$ is projected onto the line defined by $\mathbf{w}$ using the dot product: $\text{proj}_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i$
   - This gives the component of $\mathbf{x}_i$ along the direction of $\mathbf{w}$.

2. **Variance after Projection:**
   - The variance of the points after projection onto $\mathbf{w}$ is given by the standard formula for variance:
     $\text{Var}(\text{proj}_{\mathbf{w}}(\mathbf{x})) = \frac{1}{n}\sum_{i=1}^{n}(\text{proj}_{\mathbf{w}}(\mathbf{x}_i) - \text{mean}(\text{proj}_{\mathbf{w}}(\mathbf{X}))^2$
     where $\text{mean}(\text{proj}_{\mathbf{w}}(\mathbf{X}))$ is the mean of the projected points.
3. **Expressing Variance in Terms of $\mathbf{w}$:**
   - Substituting the expression for projection $\mathbf{w}^T\mathbf{x}_i$ into the variance formula, you get:
     $\text{Var}(\text{proj}_{\mathbf{w}}(\mathbf{x})) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{w}^T\mathbf{x}_i - \text{mean}(\mathbf{w}^T\mathbf{X}))^2$
4. **Maximizing Variance:**
   - The goal is to find $\mathbf{w}$ such that this variance is maximized. This involves finding the eigenvector associated with the largest eigenvalue of the covariance matrix of the data.

Simple Neural Networks-

How they work (visualization and vector)



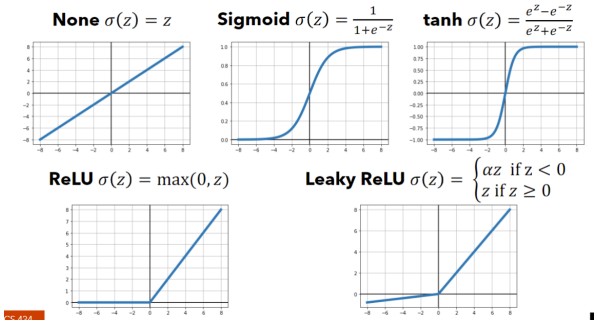
$$\underset{1\times1}{\begin{bmatrix} a \end{bmatrix}} = \sigma\left(\underset{d\times1}{\begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}^T} \underset{d\times1}{\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}} + \underset{1\times1}{\begin{bmatrix} b \end{bmatrix}}\right)$$

, where σ(z) = 1/(1+e^-z) and z = w^Tx +b (the individual neuron)

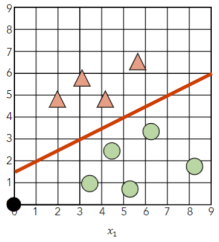
Neuron is a linear function of its input followed by a (typically) non-linear activation function to produce output.

Hyperparameters : activation function (σ)

Learnable Parameters: $\mathbf{w} \in \mathbb{R}^d$ and $\mathbf{b} \in \mathbb{R}$



Example of how w^Tx works



Step 1) Draw a line separating the classes

Step 2) Figure out the equation of that line

A $-0.5x_1 + 1x_2 - 1.5 = 0$

B $0.5x_1 - 1x_2 + 1.5 = 0$

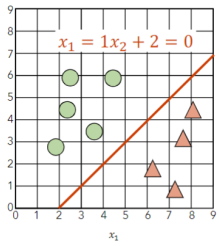
Step 3) Check the sign on each side of the line
(I usually use (0,0) such that value is just -b).

A $-0.5 * 0 + 1 * 0 - 1.5 = -1.5$

B $0.5 * 0 - 1 * 0 + 1.5 = 1.5$

Step 4) Choose weights that match pos/neg classes

$w = [0.5, -1]$ $b = 1.5$



Step 1) Draw a line separating the classes

Step 2) Figure out the equation of that line

A $-1x_1 + 1x_2 + 2 = 0$

B $1x_1 - 1x_2 - 2 = 0$

Step 3) Check the sign on each side of the line

(I usually use (0,0) such that value is just -b).

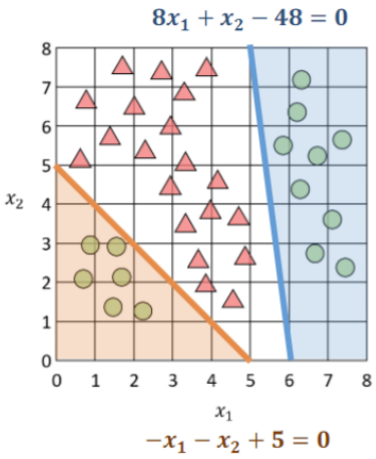
A $-1 * 0 + 1 * 0 + 2 = 2$

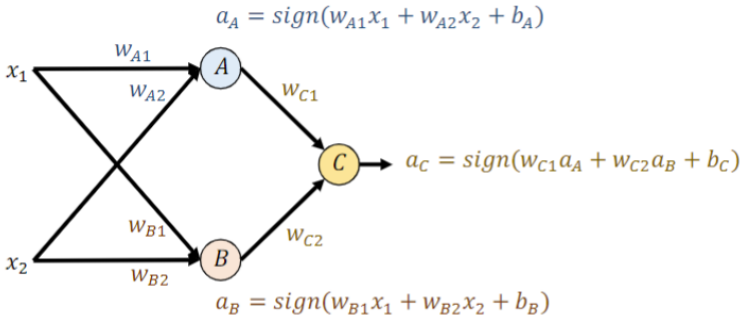
B $1 * 0 - 1 * 0 - 2 = -2$

Step 4) Choose weights that match pos/neg classe

$w = [-1, 1]\ \ b = 2$

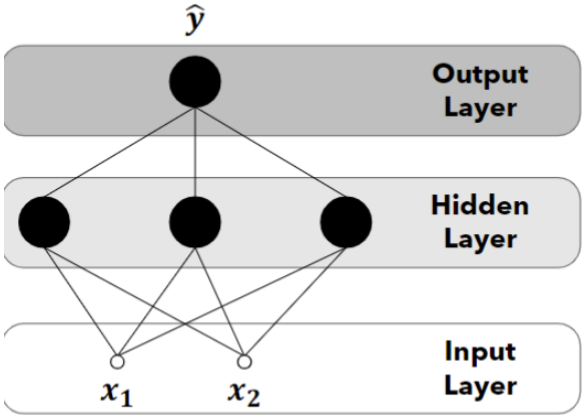
Example for combining two neurons





Overall how they work:

A Neural Network is a set of connected neurons. A very typical arrangement is a feed-forward multilayer neural network like the one shown below.



Each layer receives its input from the previous layer and forwards its output to the next – thus the **feed-forward** description.

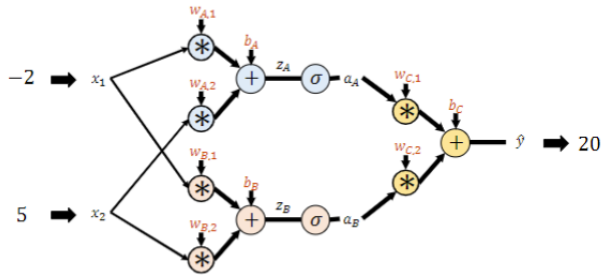
The layers of neurons between the input and output are referred to as **hidden layers**.

- This network for instance is **a 2-layer neural** network (1 hidden and 1 output)
- Number of neurons in a layer is referred to as its width.

Activation functions in different layers can be heterogenous.

- Output layer's activation is task dependent
  - Linear for regression
  - Sigmoid or softmax for classification

Loss Function *L*- A function measuring "how bad" a network's output is, usually relative to some gold-standard for what the output should be.



$\boldsymbol{x} = [-2,5]$ with y value 15

$$\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$$

For example, the L2 loss is commonly used for regression

$$\mathcal{L}(y = 15, \hat{y} = 25) = (15 - 25)^2 = 100$$

Use gradient descent to slowly reduce the size of loss.

We had an example $\boldsymbol{x} = [-2,5]$ with y=15 and our network predicted $\hat{y} = 20$.

$$\hat{y} = w_{C,1}a_A + w_{C,2}a_B + b_C$$

How should b_c change to make the prediction more correct if we reran it?

Get more negative so $\hat{y}$ reduces.

What neural networks are useful for:

Non-linear Function Learning: As we saw in the demo yesterday, neural networks can learn non-linear decision boundaries. But unlike the kernel methods / basis functions we used before, we don't need to specify the exact from of this non-linear function.

Theoretically Universal Approximators: For any continuous function F over a bounded subspace of D-dimensional space, there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximates F arbitrarily well. That is to say, for all x in the domain of F, $\left|F(x) - \hat{F}(x)\right| < \epsilon$

Incredibly Flexible: Can be used for classification or regression. Or just any problem with a differentiable loss function.

Regression Loss Functions

Squared Error

$$L(a,b) = (a-b)^2$$

Absolute Error

$$L(a,b) = |a-b|$$

Huber

$$L(a,b) = \begin{cases} \frac{1}{2}(a-b)^2 & \text{if } |a-b| < \delta \\ \delta|a-b| - \frac{1}{2}\delta^2 & \text{else} \end{cases}$$

Classification loss functions:

Cross Entropy ($\vec{a} \in \Delta^C$ $\vec{b} \in \Delta^C$)

$$L(\vec{a},\vec{b}) = -E_b[\log(a)] = -\sum_{i=0}^{C} b_i \log(a_i)$$

$$\Rightarrow \; if\ \vec{b} = \vec{1}_c \;\rightarrow\; -log(a_c)$$

Hinge ($\vec{a} \in \mathbb{R}^C$ $\vec{b} = \vec{1}_c$)

$$L(\vec{a},\vec{b}) = \sum_{i=0}^{C} \max(0, \delta - b_i a_i)$$

$$\Rightarrow \; if\ \vec{b} = \vec{1}_c \rightarrow \max(0, \delta - b_c a_c)$$

General idea behind training a neural network

Forward Pass

1) For each training example:
   - Compute and store all activations
   - Compute loss

Backward Pass

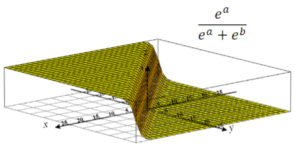
2) Compute gradient of the loss with respect to all network parameters
   - Will do this efficiently with an algorithm called **backpropagation**

Update

Take a step of gradient descent to minimize the loss

This functional form has a name – the softmax function. It turns a real-valued vector $\mathbb{R}^d$ to a categoric distribution -- i.e. each element in (0,1) and it sums to 1.

$$Softmax\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{bmatrix}\right) = \begin{bmatrix} e^{z_1} \\ e^{z_2} \\ \vdots \\ e^{z_d} \end{bmatrix} \frac{1}{\sum_{i=1}^{d} e^{z_i}}$$



"**max**" – amplifies probability of largest x_i
"**soft**" – differentiable and still assigns non-zero probability to other entries

Backpropagation – A reverse-mode automatic differentiation algorithm commonly used to efficiently compute parameter gradients when training neural networks via gradient descent.

Builds off two simple observations / ideas:

1) Neural networks tend to have lower dimensional outputs than inputs.

2) We shouldn't recompute something we already know.

Forward Mode Differentiation:

Forward mode differentiation, also known as forward-mode autodiff or forward-mode automatic differentiation, is a technique used to compute the derivative of a function by differentiating it with respect to one input variable at a time. In forward mode, the computation starts from the input variables and progresses toward the output. Intermediate values and derivatives are computed simultaneously using a process known as "dual numbers" or "dual arithmetic." This approach is particularly useful when the number of input variables is smaller than the number of output variables.

Training Neural Networks Summary:

1. **Forward Mode Differentiation:**
   - **Purpose:** Compute the derivative of a function by differentiating it with respect to one input variable at a time.
   - **Usage in Neural Networks:** Less common; suitable for scenarios where the number of input variables is small compared to the output variables.
   - **When to Use:** Useful when dealing with a few input variables and a relatively larger number of output variables.

2. **Backpropagation (Reverse Mode Differentiation):**
   - **Purpose:** Efficiently compute the gradient of the loss function with respect to network parameters (weights and biases).
   - **Usage in Neural Networks:** Widely used for training neural networks due to its efficiency.
   - **When to Use:** Preferred for neural networks with a large number of input variables and a relatively smaller number of output variables.

3. **Reverse Mode Differentiation:**
   - **Purpose:** General technique for computing derivatives of a scalar-valued function with respect to its inputs.
   - **Usage in Neural Networks:** Encompasses backpropagation; applicable in various optimization scenarios.
   - **When to Use:** Suitable for scenarios where a general approach to computing derivatives is required, not limited to neural networks.

4. **Computation Graph:**
   - **Purpose:** Graphical representation of mathematical expressions or computational processes.
   - **Usage in Neural Networks:** Essential for understanding dependencies and implementing forward and reverse mode differentiation.
   - **When to Use:** Especially useful when visualizing complex mathematical expressions and understanding the flow of computations in neural networks. A directed acyclic graph (DAG) with

   vertices corresponding to computation and edges to intermediate results of the computation.

In Summary:

In neural network training, the choice between forward mode differentiation, backpropagation (reverse mode differentiation), and computation graphs depends on the specific characteristics of the problem at hand. Consider the size of input and output variables, computational efficiency, and the need for a general approach when selecting the appropriate training technique.

1. **Feedforward Neural Network:**
   - **Neuron Input:** $z = \sum_{i=1}^{n} w_i \cdot x_i + b$
   - **Neuron Output (with Activation):** $a = \sigma(z)$, where σ is the activation function.
2. **Activation Functions:**
   - **Sigmoid Activation:** $\sigma(z) = \frac{1}{1+e^{-z}}$
   - **Hyperbolic Tangent (tanh) Activation:** $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
   - **Rectified Linear Unit (ReLU) Activation:** $f(z) = \max(0, z)$
3. **Loss Function:**
   - **Mean Squared Error (MSE):** $L(y, \hat{y}) = \frac{1}{2n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$
4. **Backpropagation:**
   - **Gradient Descent Update Rule:** $w_{ij} = w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}$, where α is the learning rate.
   - **Chain Rule for Backpropagation:** $\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$
5. **Softmax Activation (for Multiclass Classification):**
   - $P(class_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$, where C is the number of classes.