Supervised Learning

functional framework of (supervised) machine learning

- $\theta^* = argmin(\sum L[Y_i = f(X_i; \theta, \lambda)])$
  - parameters θ (eg. learning rate, batch size)
  - hyperparameters λ (eg. weight, bias)
  - linking function f
  - loss function L

1. Linear Regression

1. **predictor and outcome**
   - estimates the **regression coefficients** β_0, β_1 that map X to f(x) by **minimizing the sum of squared differences** between **predicted and actual outcome (residuals)**
     - **linear** in the **parameters**, not necessarily variable
2. **pros and cons**
   - pros: fast, scales well, low risk of overfitting
   - cons: tendency to underfit, needs lot data if many predictors, many situation not linear, no causality

Assumption	Mathematical reason for it	Issue if violated	Fix?
Linearity	$Y=X\beta+\varepsilon$	Model systematically wrong (curved residuals)	**Common**: Transforms (e.g. log, square, etc.) **Better**: Use a nonlinear model
Independence of errors	$E[\varepsilon_i\varepsilon_j] = 0$ (independent observations)	True variance is under-estimated (standard error and CI estimates are off)	Use time-series models like ARIMA Use clustered standard errors (appropriateness depends on use-case/situation)
Homoscedasticity	$Var(\varepsilon_i)=\sigma^2$ Constant variance	Predictions less precise where variance is high	Just inference: Heteroscedasticity-Robust Standard Errors Model coefficients matter: Weighted least squares (WLS)
Normality of errors	$\varepsilon \sim N(0,\sigma^2)$	Hypothesis tests and CIs can be misleading	**Common**: Bootstrapping **Better**: Robust regression models
No perfect multicollinearity	$(X^TX)^{-1}$ must exist	Unstable β coefficients	**Common**: Remove redundant variables **Better**: Regularized regression

3. **Interpret model coefficients**

Outcomes — How much of this variable there is

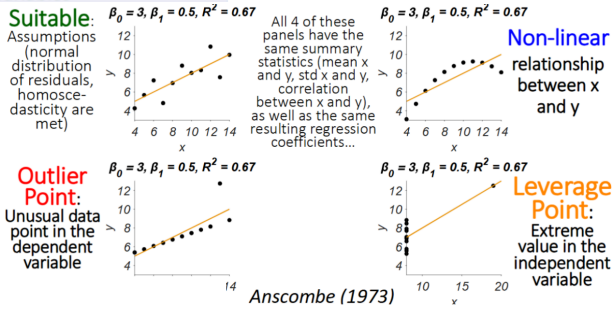
$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$$

Offset — How much it matters — "Error"

4. **expansion models**
   1. **multiple regression**
      - adding predictors increases the variance of the outcomes that is explained of a regression model
   2. **polynomial regression**

$$y = \beta_0 + \beta_1 x_1 + \beta_2 {x_2}^2 + \beta_3 {x_3}^3 + \cdots + \beta_n {x_n}^n + \epsilon$$

      - still linear regression since linear with respect to parameters, though nonlinear with x



5. **assessment**
   - **RMSE** = $\sqrt{\frac{\sum(\hat{y}-y)^2}{n}}$, the SD of the residuals
   - $COD/R^2 = \frac{SS_{explained}}{SS_{explained}+SS_{residual}}$: The squared multiple correlation. Proportion of the variance that can be accounted for by the model. Ranges between 0 and 1, negative indicates a model is systematically guessing worse than the mean of the outcomes
   - R: the multiple correlation /pearson correlation. Correlation between $\hat{Y}$ and y. 1: no regression to the mean effect. 0: strongest regression to the mean effect, no linear relationship (but maybe nonlinear)
6. **issues** when adding predictors
   - **overfitting**
     - fix: **cross-validation** into training set and test set
       - if each set not independent, **leakage occurs**
   - **collinearity**
     - problem: 2 predictors that are each highly correlated
7. **fix by Regulation: Regularized Regression**
   - **Bias / variance tradeoffs**
     - **high bias**: function too simple (less variable), underfit, model doesn't fit well
     - **high variance**: function too flexible (much variable), overfit, fit doesn't transfer to other dataset well

Linear regression pushes models this way →

High bias* Low variance**	Slight bias Slight variance	Low bias High variance

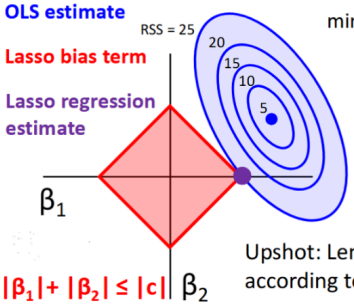
1. **Ridge Regression**: **add a regularization term** biases the estimator a litter, but reduces the variance a lot

L2: Penalize on Absolute value

2. **LASSO (L1 regularization)**

L1: Penalize on squared value

L1: Penalize on squared value



$\min \|\mathbf{y} - X\boldsymbol{\beta}\|^2$ subject to $\|\boldsymbol{\beta}\|_1 \le c$

Lasso regression loss function:

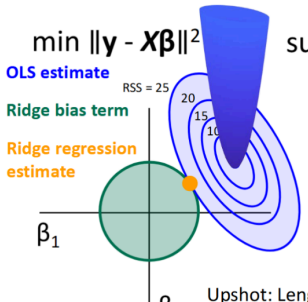
$$\min(\|\mathbf{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|_1)$$

$\boldsymbol{\beta}_{LASSO}$ = *from numerical methods* (e.g. coordinate descent, or least angle regression (LARS))

Upshot: Length of β is shorter than some constraint according to L_1 norm, by setting individual β_i to zero.

- λ increases, the radius of the constraint diamond shrinks, setting some β to 0, while shrinking others slightly

Ridge regression (L2 regularization) c = some number which serves as a constraint



$\min \|\mathbf{y} - X\boldsymbol{\beta}\|^2$ subject to $(\|\boldsymbol{\beta}\|_2)^2 \le c^2$

$\beta_1^2 + \beta_2^2 \le c^2$

Ridge regression loss function:

$$\|\mathbf{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2$$

Objective:

$$\min(\|\mathbf{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2)$$

$$\boldsymbol{\beta}_{ridge} = (X^T X + \lambda I)^{-1} X^T \mathbf{y}$$

$$\boldsymbol{\beta}_{OLS} = (X^T X)^{-1} X^T \mathbf{y}$$

Upshot: Length of β is shorter than a constraint from the L_2 norm, making all β_i smaller (but never zero).

2. Logistic Regression

- Mapping continuous inputs to **discrete outcomes**, while using a **nonlinear linking function (cross entropy loss)**
- Logistic regression gives **odds** that an outcome happens compared to it not happening, for a given predictor value
  - **Odds**: Probability of an event occurring divided by the probability of the event not occurring $\frac{p}{1-p}$
  - **Logit Function**: Links the values in the predictor variable to the probabilities of the outcomes.
  - **Logistic Function / Sigmoidal Function**: inverse logit function
    - why inverse: On logit function, probability is on x-axis; inverse make probability on y-axis
- **metrics**
  - **accuracy** is not enough: most real datasets imbalanced
  - **AUC (area under ROC curve) / AUROC**
    - **ROC (Receiver Operating Characteristic)**
- **AUC (Area under the ROC curve)**: assess the performance of classification models
  - plotting **sensitivity (true positive rate)** as a function of **1-specificity (false positive rate)**
  - it captures the classification accuracy of an algorithm for the entire range of different criteria (threshol
  - 1.0: perfect classification; 0.5: random performance, isn't learning any useful signal from data
- **the precision recall (PR) curve**
  - use case: **imbalanced dataset, focus on positive**

In stands for "logarithmus naturalis" x = Data Here: GPA score

$$\log it(p) = \ln(odds) = \ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1$$

$$\log it^{-1}(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

$$p = \frac{e^{\beta_0+\beta_1 x_1}}{(1+e^{\beta_0+\beta_1 x_1})}$$

ROC: x: False positive rate (1 – Specificity)
y: True positive rate (Sensitivity)

$Prevalence = \frac{AP}{AP+AN}$

$Accuracy = \frac{TP+TN}{AP+AN}$

$Sensitivity = \frac{TP}{AP} = \frac{TP}{TP+FN}$ aka *Recall* aka *Power* aka *True Positive Rate*

$Specificity = \frac{TN}{AN} = \frac{TN}{TN+FP}$ aka *Selectivity* aka *True Negative Rate*

$Precision = \frac{TP}{PP} = \frac{TP}{TP+FP}$ aka *Positive Predictive Value* aka *PPV*

$NPV = \frac{TN}{PN} = \frac{TN}{TN+FN}$ *Negative Predictive Value*

Ground truth	+ (PP) Predicted signal	- (PN)
Actual signal + (AP)	True positive (TP) "Hit" Correct classification	False negative (FN) "Miss" β error / Type II error
Actual signal - (AN)	False positive (FP) "False alarm" α error / Type I error	True negative (TN) "Correct rejection" Correct classification

What does it mean?

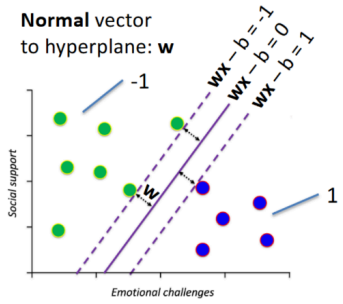
Sensitivity: Ability to detect real signal

Specificity: Ability to not hallucinate

Precision: "I called it". Did you?

3. Support Vector Machines

- **basic idea:** use **maximal margin classifiers** to pass a **linear separable hyperplane** through data to classify it into 2 groups s.t. the **margin is maximized**
- **find optimal hyperplane efficiently**



Hyperplane: **wx** - b = y

y = 0: An outcome on the decision boundary

We start with **x** on the decision boundary, so **wx** - b = 0

How many unit vector steps c in direction of w until we hit an edge? $\mathbf{w}\left(\mathbf{x} + c\frac{\mathbf{w}}{\|\mathbf{w}\|}\right) - b = 1$

$\mathbf{wx} + c\frac{\mathbf{ww}}{\|\mathbf{w}\|} - b = 1$ $\quad \mathbf{wx} - b = 0$ $\quad \mathbf{w}^T\mathbf{w} = \|\mathbf{w}\|^2$

$c\frac{\|\mathbf{w}\|^2}{\|\mathbf{w}\|} = 1$ $\quad c\|\mathbf{w}\|^2 = \|\mathbf{w}\|$ $\quad c = \frac{1}{\|\mathbf{w}\|}$ Margin size: $\frac{2}{\|\mathbf{w}\|}$

Labels, class membership, y: {-1, 1}

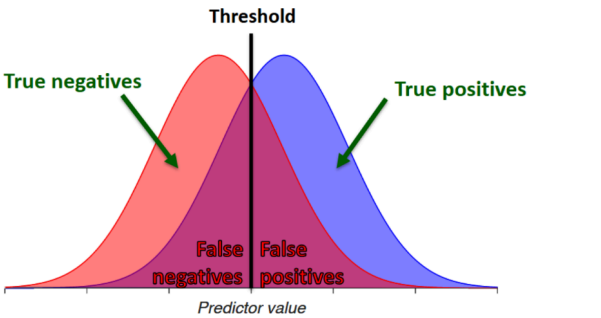
w: **Weights** b: **Bias** x: Inputs, Predictors

Looking for: w and b that maximize margin

To maximize the margin, we have to minimize $\|\mathbf{w}\|$, so that **y**(**wx**-b) ≥ 1

- **why maximize margin**: more stable, not easy to overfit to new sample

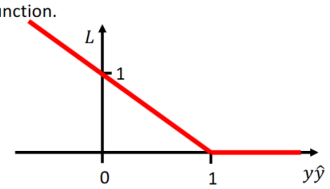
- **Loss function: Hinge Loss**
  - **If predict wrong,** $y_i(\mathbf{w}^T x_i + b) \le 0$
    - L will be greater than 1, large loss
  - **If predict correct, but outside the margin (high confidence),** $y_i(\mathbf{w}^T x_i + b) \ge 1$
    - L will be 0. No loss on these points
  - **If predict correct, but inside the margin (low confidence),** $0 \le y_i(\mathbf{w}^T x_i + b) \le 1$
    - L will be between 0 and 1, but too close to the decision boundary, so still some loss
- **soft margin classifiers**: allowing for misclassifications in the margins of the training set



- Conceptually, hinge loss introduces the concept of a penalty that is incurred, if the distance from the decision boundary is not large enough, even if the observation is technically classified correctly.
- Which makes for a more gradual loss function.

$$L = \max(0, 1 - yi\ (\mathbf{w}^T\mathbf{x}_i + b))$$

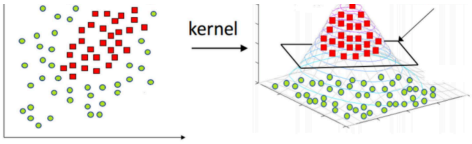
True class label (-1 or 1) — Predicted class score $\hat{y}_i$



- **SVM kernels tricks**

use case: For non-linear classification problem

Intuition: transform original space into a **higher dimensional space** by creating synthetic features with kernel functic (not mapping like neural network) so that the data will be **linearly separable** in higher dimensional space



4. Decision Tree

- **Why tree**: Allow for the classification of highly complex, linearly non-separable dataset; Intuitive, Interpretable
- splitting boundary / quantify leaf impurity
  - **Gini Impurity**
    - $GI = 1 - \sum_{i=1}^{n} p_i^2$
    - p_i: The probability of class i in this leaf node
    - **Goal**: find variable with the smallest **GI** (pure leaf), the more likely it can classify samples
  - **Entropy**
    - $H(x) = -\sum p(x) log p(x)$
    - **Goal**: find variable with the smallest **Entropy** (pure leaf), the more likely it can classify samples
    - **Entropy has larger penalty to mixed sample to maximize information gain from the split**

Revisiting Diathesis: Calculating node impurity

GI = $1 - p^2 - (1 - p)^2$ H = $-p \log_2(p) - (1-p) \log_2(1-p)$

Diathesis? H: D? Y 4 N 2; L: D? Y 1 N 5

$GI_{4,2} = 1 - \left(\frac{4}{6}\right)^2 - \left(\frac{2}{6}\right)^2$

$GI_{4,2} = 1 - \frac{16}{36} - \frac{4}{36}$

$GI_{4,2} = \frac{16}{36} \approx 0.444$

$GI_{1,5} = 1 - \left(\frac{1}{6}\right)^2 - \left(\frac{5}{6}\right)^2$

$GI_{1,5} = \frac{10}{36} \approx 0.278$

$GI = \frac{6 * 0.444 + 6 * 0.278}{12} \approx 0.361$

$H_{4,2} = -\frac{4}{6}\log_2\left(\frac{4}{6}\right) - \frac{2}{6}\log_2\left(\frac{2}{6}\right)$

$H_{4,2} \approx 0.918$

$H_{1,5} = -\frac{1}{6}\log_2\left(\frac{1}{6}\right) - \frac{5}{6}\log_2\left(\frac{5}{6}\right)$

$H_{1,5} \approx 0.65$

$H = \frac{6 * 0.918 + 6 * 0.65}{12} \approx 0.784$

- **Ensembles** to improve performance
  - **why: single tree is weak learner due to high variance (easy to overfit)**
    - **strong learner**: get the model error below some small threshold with high probability
  - **methods**
    - **bagging (bootstrap aggregating)**
      - **random forest**: random sampling from sample to train multiple trees, then use voting or average result to reduce variance
    - **boosting**
      - first tree learns first, second tree learn what first tree classify wrong to reduce bias
      - **AdaBoost**: forest of stumps
        1. first stamp, set weights of each sample to $\frac{1}{N}$

           recurrence:
        2. pick variable minimize Gini (maximize classification purity)
        3. The error is the sum of the weights of incorrectly classified cases
        4. Use the error to determine the **voting power** of the stump, $vp = \alpha = \frac{1}{2} ln(\frac{1-e}{e})$
        5. update the weights for the next stump, and normalize
           - $w_{new-correct} = w_{old-correct} * e^{-\alpha}$: decrease weight
           - $w_{new-incorrect} = w_{old-incorrect} * e^{\alpha}$: increase weight
      - **Gradient Boosting**

        Recurence:
        1. calculate the log of odds for instance, then convert to probability. $p(depressed) = \frac{e^{ln(odds)}}{1+e^{ln(odds)}}$
        2. calculate **residues** of each leaf between predictions and actual outcomes
        3. take the gradient of the loss and step in the negative direction
        4. update: $new\ ln(odds) = old\ ln(odds) + lr * tree\ prediction$

- Extracted features in the hidden layer are usually meaningless (not interpretable)

CNN

- why CNN: In fully connected layers, hidden layer activations are hard to interpret. CNN take a neighboring pixels
- **kernel**: filter, weights we train
- **feature map**: convolved image after applying filter and bias
- **problem with deep network**
  - **vanishing gradient**: component derivatives are < 1
  - **exploding gradient**: component derivatives are > 1

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,n} \\ w_{3,1} & w_{3,2} & w_{3,3} & \cdots & w_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{k,1} & w_{k,2} & w_{k,3} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_k \end{bmatrix}$$

15x25 25x1 15x1

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{a} + \mathbf{b})$$

15x1

Neural Network

- #parameters =

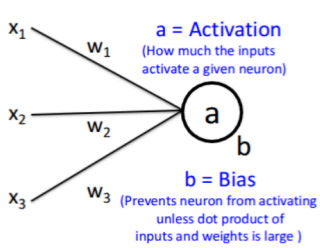
$$input\ neurons \times hidden\ neurons + hidden\ neurons \times output\ neurons + hidden\ neurons + output\ neurons$$

Structure

- An artificial neural network is a function approximator:

Labels ↓ Inputs

- y = f(x)



a = Activation (How much the inputs activate a given neuron)

$\mathbf{a} = \sigma(\mathbf{\Sigma}\ \mathbf{wx} + \mathbf{b})$

Σ wx = Dot product of inputs and weights (Picking up on a specific pattern in the inputs)

σ = Activation function (Nonlinear, like sigmoid or reLU, allows to approximate nonlinear functions)

b = Bias (Prevents neuron from activating unless dot product of inputs and weights is large)

*: Assuming
-Infinite # of neurons
-Infinite amounts of data
-Clean/good data
-Infinite time/energy to train

Learning: Adjusting the weights and biases to minimize the output of the cost function

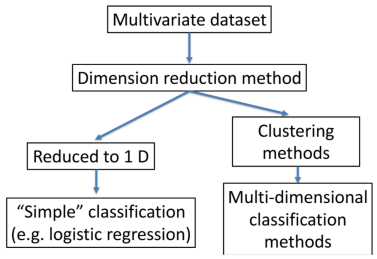
- **Cost function**: only one number: Average cost over all training data
- **Adjusting: backpropagation**: -delta to lower cost

• Gradient descent is an iterative algorithm:

1. Determine the loss function (which yields the format of the gradient)
2. Initialize the parameters with arbitrary (usually random) values
3. Compute the gradient using these parameter values
4. Take a **step** in the direction of the gradient (add or subtract the gradient value from/to the respective parameter), scaled by the **learning rate**
5. Go back to 3), until the step size is under some threshold, then terminate

Unsupervised Learning

process



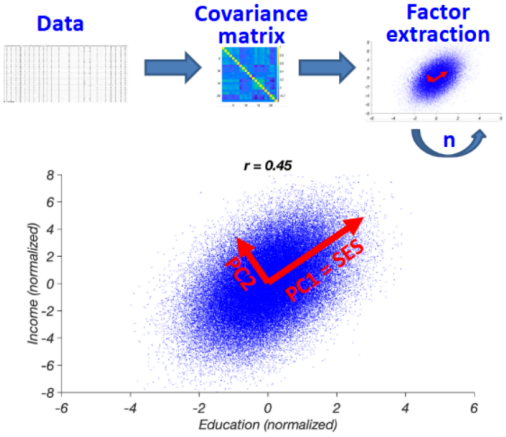
Dimensionality Reduction

- **motivation**: too much going on on multi dimension, correlated information is redundant

Linear

1. Principal Component Analysis (PCA)

- PCA finds an orthonormal basis and projects the data onto this basis.
- It does so by computing the covariance matrix.
- The Eigenvectors of this covariance matrix are the basis vectors of this orthonormal basis.
- The Eigenvalues of these Eigenvectors correspond to how much of the variance in the data is accounted for by the respective vector.
- PCA finds the Eigenvector that – if projected onto – the variance of the projected data is maximal (has the largest Eigenvalue).
- This Eigenvector is then called the first principal component.
- The data is then projected onto the second – orthogonal – Eigenvector of the data that has – if projected – the second highest variance (which becomes the 2nd principal component).
- For a dataset with n variables, this process is repeated n times, yielding n principal components.



- **steps**
  1. start with dataset with many correlated features
  2. standardize the data
  3. compute the covariance matrix
  4. Find the eigenvectors of the covariance matrix
  5. project the data on these principal components (PCs)
  6. retain as many PCs as needed to retain an acceptable proportion of variance
- **remark**
  - focus on direction **maximizing variance**, not direction helping classification
  - works well when class labels are missing (unsupervised learning)
  - If know class labels, PCA might through away factors helpful for classification

2. Linear Discriminant Analysis (LDA, supervised)

- project into a dimension that **maximize the distance between projected class means**, **minimizes the variance within the projected classes**
  - require class labels, assumes gaussian distribution of data within classes
- **steps**
  1. calculate the covariance matrix for each class, take the weighed average of each matrix to get W
  2. compute the total covariance of the data regardless of class T, subtract it by the pooled within class covariance W to get B: B = T - B
  3. $S = W^{-1}B$
  4. compute the eigenvector of S: [a_1, a_2, a_3 ....]
  5. each column $L_1 = a_1X = a_{11}X_1 + a_{12}X_2 \ldots$, $LDA = [L_1, L_2, \ldots]$
  6. now data has been extracted to LDA dimension, you can train model with it

How does LDA find the orientation of this dimension?

$$L = a_1X_1 + a_2X_2 + a_3X_3 + \ldots a_nX_n$$

such that the scores of the compound variable L maximize the separability of the classes:

$$Separation = \frac{Between\ class\ mean\ distances}{Sum\ of\ the\ variance\ within\ classes}$$

Eigenvectors of S yield the weights a_i $\quad \boldsymbol{S = W^{-1}B}$ B: Between class cov matrix W: Within class cov matrix

Nonlinear

3. Multidimensional Scaling (MDS)

- can have no variables, data are the distance matrix between samples
- **purpose**: dimension itself contains dimension information. Reconstruct map with MDS low dimension matrix to approximate input dimension matrix
  - **downsides**
    - Extreme computational complexity
    - might unable to preserve distances in the lower dimensional space
- **steps**
  1. transform data into a distance matrix $D \in \mathbb{R}^{n \times n}$
  - We then place the same data points in a lower dimensional space (initialized at random positions)
  - We then compute the "stress" as

$$Stress = \sqrt{\frac{\sum (d_{ij} - \hat{d}_{ij})^2}{\sum d_{ij}^2}}$$

where d_{ij} is the distance between any two data points i and j in the higher dimensional space and $\hat{d}_{ij}$ is the distance between the same two data points i and j in the lower dimensional space

  - We then iteratively move the points in the lower dimensional space in a way that reduces stress.
  - We keep doing this until the stress (which serves as our loss function) drops under some threshold (ideally 0)
  3. We get $Y \in \mathbb{R}^{n \times k}$ where each row represents the coordinate of a sample

4. Stochastic Neighbor Embeddings (SNE

$$L = \sum_{i,j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

p_{ij}: High dimensional affinity

q_{ij}: Low dimensional affinity

- **address the downsides of MDS**
- **preserve the nearest neighbors** (locality) in lower dimensional space, not distances,
- **steps**
  1. compute high dimensional affinity matrix P
  2. Initialize low dimension coordinates for each sample (randomly or use PCA)
  3. compute low dimensional affinity matrix Q
  4. minimize loss L (KL-Divergence) to make Q approximate P
- **parameter**
  - **perplexity**: effective width of neighborhood
- **tSNE**: use t distribution in low dimension instead of Gaussian distribution for better clustering
- **downsides**
  - still a bit slow, though faster than MDS
  - preserves the local, but not the global data structure well

5. Uniform Manifold Approximation and Projection (UMAP)

- Similar to SNE: but not using probability distribution, instead use graph
- **steps**:
  1. compute fuzzy simplicial set for each sample with k neighbors for high dimension
  2. Initialize low dimension coordinates for each sample (randomly or use PCA)
  3. compute fuzzy simplicial set for each sample with k neighbors for low dimension
  4. minimize loss to make low dimension representation approximate high dimension representation
- **pros**
  - preserves good overall structure
  - faster then t-SNE
- **cons**
  - sensitive to noise and extreme values (though SNE is more sensitive)
  - large number of hyperparameters

Clustering

- **Intuition**: members of a cluster are similar, but dissimilar from members of other clusters

1. k-means clustering

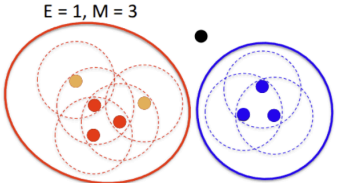
- difference with KNN: KNN has labeled data
- **process**
  1. Placing K cluster centers (centroids) at some initial location in the data space
  2. Identifying which centroid a given datapoint is closest to, for all datapoints
  3. Calculating the distance of all data points from their nearest cluster center (centroid)
  4. Summing all of these distances
  5. Adjusting the location of the centroids such that the total distance of all points from their centroids decreases.
  6. Iteratively doing that until the total distance / centroid location doesn't change anymore
- **parameters**
  - **K**: the number of clusters
  - Initial centroid locations (default: random)
  - Distance metric (Euclidian distance $||x||_2 = \sqrt{\sum_i x_i^2}$)
- **pros**:
  - yield optimal location of cluster centers that minimize the total distance of all data to centers
  - labeling the data in terms of their nearest center
- **cons**:
  - usually converge, but may not converge if the cluster centers start unevenly (eg. all in the same position)
  - it does not yield the optimal number of clusters, only optimal location of cluster centers
  - **solution**:
    1. the elbow method: find variance within cluster for each K, if it starts becoming flat at k, then k is elbow point
    2. **the sihouette method**
       - **process**
         1. Apply the K-means clustering method for a range of plausible values K (e.g. 2-10)
         2. For each of these –and each data point i-calculate the **Silhouette Coefficient** s(i):
            1. calculate the average distance of a point i to all other points in the same cluster as a(i)
            2. Then calculate the average distance of a data point i to all points in the nearest cluster as b(i)
            3. Then calculate s(i) as (b(i)–a(i) / max([b(i),a(i)]))
         3. s(i) will vary from 0 (if classification is arbitrary) to 1 (if classification is ideal). Negative s(i) values indicate misclassification.
         4. Make a histogram of the s(i) values for all values of K that you tried
         5. choose the k (number of clusters) with the highest sihouette score
       - **cons**: yields spherical clusters, sensitive to outliers

2. GMM

- difference between k-means: k-means is hard clustering, where each point is assigned to only one cluster center, and assumes each cluster comes from an isotropic gaussian distribution; GMM assumes data is the mixture of different gaussian distribution, so each point has its probability distribution on each cluster, and covariance matrix makes ellipse clusters.

3. DBSCAN

- density-based spatial clustering of application with noise
- can find clusters with non-regular (non-spherical, non-convex) shapes
- **idea**: Use **Density** (number of data points per area) instead of distance
- **parameter**
  - **Epsilon**: The radius around a data point that forms a region/neighborhood within the perimeter
  - **MinPoints**: The minimum number of data points (including starting point) that must be in the region (within the perimeter) for a cluster to be formed
- **steps**
  - •Data points that are cluster members "broadcast" their own perimeter to look for new cluster members.
  - •Based on the density concept (implemented via epsilon [E] and minPoints [M]), DBSCAN distinguishes 3 kinds of data points:
  - •"Core point": Has N >= M points within E
  - •"Border point": N < M, but reachable from core point
  - •"Outlier point": Not a core point, nor reachable



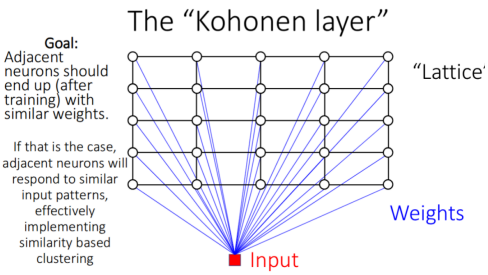
pros and cons

- pros
  - finds the number of clusters
  - yiled clusters of arbitrary shape
  - not sensitive to outliers
  - does not include noise in the clusters
- cons
  - slow
  - relies on sharp density drops to identify cluster boundaries
  - determining epsilon and minPoints can be arbitrary

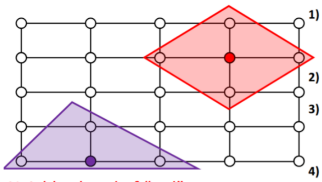
Autoencoder

- **Loss function measure reconstruction error (same input output size)**
- **Dimensionality reduction here to find latent space that represents the structure of data in fewer dimesions**

Self Organizing Map



Mechanism that maps similar input patterns to neighboring regions of the lattice:
Competition and collaboration in Kohonen networks



1) **Competition**: Identify the neuron with the weight vector that is most *similar* to the input data. This is the "winning neuron" or "best matching unit" (BMU)
2) **Collaboration**: Identify the neurons that are in the neighborhood of the BMU.
3) **Learning**: The weights of the neurons in the neighborhood are updated such that they become *more* similar to the input data.
4) **Iteration**: Steps 1-3 are repeated until some number of iterations is reached, or the weights don't change any more.

How does the weight updating work, without backpropagation?

1) Find the winning neuron i: max ($\mathbf{w}_i^T\mathbf{x}$)
2) Determine which other neurons are in the neighborhood of i, usually by some adaptive (initially wider, then more narrow) gaussian "neighborhood function" h_{ij} that uses distance (from i) and time (iteration) to determine the neighborhood radius.
3) Update the weights of all neurons j in the neighborhood of the winning neuron (only!) like this:

$$w_j(t+1) = w_j(t) + \Delta w_j$$

$$\Delta w_j = \eta(t) h_{ij}(t)(X - w_j)$$

w_j: Weight of neuron j
t: Iteration
η: Learning rate
h_{ij}: Neighborhood function
X: The input vector (data)
$X - w_j$: "Displacement vector"

- Kohonen networks / SOM can be used for clustering.
- The idea is to label the data (once all the neuron weights have converged) by the neuron that is closest to them.
- Functionally, the solution of Kohonen networks / SOM closely corresponds to the kMeans solution.

Reinforcement Learning

- reward r_t: scaler feedback signal that denotes how the agent is doing at time t
- **Reward Hypothesis**: The agent's goal is to **maximize expected cumulative reward**
- **States** s: The cells in the grid the robot can occupy
- **Actions** a: A set of operations that allow robot to transition from state to state
- **Immediate reward**: cost of an action. eg. -1
- **Future reward**: The value obtained by reaching a terminal cell

Policy in **Markov Decision Processes (MDP)**

- **state fully observable (no hidden state)**
- **not path-dependent**: Future not dependent on the past, only depends on current stat
- **All relevant info of past is contained in current state s**
- **policy maps state to an (optimal) action**

Bellman equation (Q-Learning)

It integrates immediate and future rewards, allowing to iteratively set the Q scores:

Here: $r = -1$ Discount factor

$$Q(s,a) = r + \gamma \max Q(s',a')$$

Value — Immediate reward — Here: $\gamma = 1$ — Future (neighboring) reward

- **Steps**
  1. Initialize Q(s, a) arbitrarily
  2. Observe current state `s`
  3. Choose action `a` (e.g., ε-greedy)
  4. Observe reward `r` and new state `s'`
  5. Update Quality Q

Issues

1. **The explore/exploit tradeoff**
   - exploit: choose best-known action
   - explore: Try new action to discover better paths
   - **solution:** represents options as samples of probability distributions. As gain more confidence about the parameters of distributions, choose random options less often
2. **Rewards are sparse**
   - then not sufficient to learn optimal behavior
   - **solution**: reward exploration itself
3. **Rewards validity**
   - What is learned with this reward
4. **Memorization vs Learning**
5. **Scores are essential**
   - If score not available, and we use human judgment, how to get super-human performance that way?
6. **Large state spaces**
   - small robot, big world --> not remotely tractable some move
7. **Delayed feedback -> the credit assignment problem**
   - don't know which move is winning / losing move
   - **root cause**: Rewards/outcomes are much less frequent than actions
   - **solution**: **Reward shaping**: Give partial credit

Deep Q Network

- **Training Steps**
  1. Input: State `s`
  1. Output: Q-values for each possible action
  1. Choose action using ε-greedy
  1. Store experience `(s, a, r, s')` in **replay memory**
  1. Train network using batches sampled from replay memory
- **Replay Memory**
  - state sequences are often highly correlated as agent explores environment, which would not be efficient
  - replay memory is a buffer containing last N experiences
  - prevent correlated state sequences from destabilizing learning
  - encourage learning from diverse, uncorrelated experience