Evaluation

True positive rate, TPrate = TP/P;

False positive rate, FPrate = FP/N; true positive = guess

- Sensitivity = TPrate = TP/P;
- Specificity = TN/N = 1 - FPrate.

- Definitions from pattern recognition:
  - precision = TP/P';
  - recall = TP/P;
  - accuracy = (TP+TN)/(P+N).

		actual value		
		p	n	total
prediction outcome	p'	True Positive	False Positive	P'
	n'	False Negative	True Negative	N'
	total	P	N	

the above is a confusion matrix
sensitivity – TP/(TP+FN) also called recall
specificity – TN/(TN+FP)
precision – TP/(TP+FP)
high sensitivity means few FN, low many FN
ROC- receiver operating characteristic
ROC space – precision/recall curve
Area under ROC curve roughly equal to the probability of classifying a positive same as positive (TP rate)
ROC curve plots TP rate (sensitivity) on the y-axis and FP rate on the x-axis
To make ROC curve first you predict your data then you compute the FP/TP rate using predicted labels comparted to actual labels, connect all the dots
Accuracy of A can be estimated as the area under the curve (if curve is diagonal accuracy of 50%)
Association Rules – want to identify correlations btw data (does A→B)

Support = % of transactions (buskets) containing $A\&B$ together.

Confidence = % transactions that have A have B as well.

$Supp(A \Rightarrow B) = \text{prob}(A \wedge B)$

$Conf(A \Rightarrow B) = \text{prob}(B|A) =$

support – do they appear together
confidence – if one appears does it imply the other
Anti-monotonic property – if an item set is not frequent all of its supersets cannot be frequent OR if an item set is frequent all of its subsets cannot be frequent
Apriori Algorithm

- Find all frequent itemsets (above min support level)
- Join step (if OP, OM are frequent check OPM most likely to be frequent)

Order matters in AR mining 1→3 != 3→1

A working definition: Given a dataset $D = \{x_1, x_2, \cdots, x_n\}$, a similarity measure $\text{sim}(x_i, x_j)$ for $x_i, x_j \in D$, and an integer k, the clustering problem is to define a mapping $f : D \mapsto \{1, 2, \cdots, k\}$, so that

- each x_i is assigned to a cluster $C_l, l \in \{1, 2, \cdots, k\}$;
- $\text{sim}(x_u, x_v) > \text{sim}(x_u, x_w)$, for any x_u and x_v belong to the same cluster and any x_u and x_w belong to different clusters.

Clustering
Distance between clusters

- **Single link** – distance btw closest elements in cluster
- **Complete link** – distance btw furthest elements in cluster
- **Average link** – average across all pairwise combinations of elements in cluster
- **Centroid** – distance btw centers of clusters

Agglomerative clustering

- Start with each data point as a cluster
- Repeatedly combine two closest clusters into one
- Until form one cluster
- Time O(kn^2) k num clusters
- Space O(n^2)

Divisive clustering

- Start with whole dataset as one cluster
- Repeatedly split until each cluster has one data point
- Space – O(n^2)

Partitioning clustering

- Create clusters in one step
- Need to know num clusters (k) $\sum_{l=1}^{k} \sum_{x_i \in C_l} \text{dist}(C_k, x_i)^2.$ square error metric

K Means

- Randomly pick k objects from D as representatives of k clusters
- Update class memberships – assign each object to its closest class
- Recompute the k centroid (use average of all values of k)
- Repeat until no object changes its class membership

EM Algorithm
Mixture models, each cluster corresponds to a probability
For EM algorithm want to find parameters (mean, covariance)
Have a certain number of points (suspect they came from k-different Gaussian distributions)

- Start by placing k random Gaussians
- for each point P(B|x_i) calculate the probability that point came from distribution B
- Once you have computed assignments use those numbers to reestimate mean and variances
- Iterate until convergence

E-step: perform probabilistic assignments of each data point to some class based on the current hypothesis *h* for the distributional class parameters;
M-step: update the hypothesis *h* for the distributional class parameters based on the new data assignments.
Probabilistic model-based clustering

- In this problem, we assume that the model of a system consists of a set of probabilistic distributions

$$f(x) = \sum_{i=1}^{k} \pi_i f_i(x; \theta_i)$$

where f_i is the distribution of a component, θ_i is the set of its parameters, and π_i is the weight or percentage of this component in the overall system.

want to estimate model parameters and component weights, which can be viewed as class memberships
Principal Component Analysis
Want to map data to a lower dimensional space, where patterns in the data emerge and composite features can be identified

- Adjust (linearly) the view point so that the direction of the major variance becomes one of the coordinates
- PC's are coordinates along which the data vary the most
- Compress data by taking fewer coordinates (less data but may lose info)

Want to describe original data in a new basis (eigenvector space)
Do this by finding the eigenvectors and eigenvalues of a transformation matrix

If $Vx = \lambda x$, then column vector x is an Eigenvector and real number λ is an Eigenvalue of V, respectively.

Lambda is the "stretching factor" here

PCA is a basis transformation to diagonalize the covariance matrix of centralized data $x_k, k = 1, 2, \cdots, n, x_k \in \Re^p, \sum_{k=1}^{n} x_k = 0$, defined as

$$C = \frac{1}{n-1} \sum_{k=1}^{n} x_k x_k^T = \frac{1}{n-1} X^T X = V \Sigma V^T,$$

where $X = (x_1, x_2, \cdots x_n), V = (v_1, v_2, \cdots, v_n), \lambda = \text{diag}(\lambda_1, \lambda_2, \cdots, \lambda_n)$, and v_i and $\lambda_i, i = 1, 2, \cdots, n$, are Eigenvectors and Eigenvalues of C. The new coordinates in the Eigenvector basis, i.e., the orthogonal projections onto the Eigenvectors, are called *principal components*.

Theorem: A symmetric matrix is diagonalized by a matrix of its orthonormal eigenvectors
Covariance matrix is always symmetric
PCA Procedure
Subtract the mean of each feature (dimension) must 0 center
Compute the covariance matrix
Compute eigenvectors/eigenvalues of the covariance matrix C
Create feature vector made of m eigenvectors
Here begin finding projections of each data point to transform it to a lower dim

- Information loss due to taking fewer features -
  - In essence, Eigenvalue λ_i captures the amount of variance of the data along the dimension of i-th feature.
  - So if we remove the i-th feature, we lose $\frac{\lambda_i}{\sum_{i=1}^{n} \lambda_i}$ amount of information.

- Deriving new dataset:
  FinalData = RowFeatureVector × RowDataAdjusted
  where RowFeatureVector = $(\text{FeatureVector})^T = V^T$, and
  RowDataAdjusted = mean value extracted Raw data,
  where Raw data are data points arranged in column.
  This also means that the transformation matrix is V^T: The first row of V^T is the first PC, so the first element of x' in the "PCA space", derived from transforming a data point x: $x' = V^T x$, is the mapping of x on the first PC.

- Getting old data back:

$$\begin{aligned}\text{RowDataAdjusted} &= \text{RowFeatureVector}^{-1} \times \text{FinalData} \\ &= \text{RowFeatureVector}^{T} \times \text{FinalData.} \\ &= V \times \text{FinalData.}\end{aligned}$$

  (because Eigenvectors are orthognal and normalized to have unit length)

PCA achieves

- Dimension reduction: finds an embedding of data in lower dimensional space to reduce storage/comm cost
- Remove possible noise (could remove impt info)
- Matrix factorization: general paradigm for data analysis

Data compression and noise removal, factor analysis, anamoly and outlier detection, data viz
PCA Fails – covariance extremely sensitive to large values
If each dimension is not normalized to zero mean and unit variance
Assumes the underlying subspace is linear (if it data is low dimensional is non linear PCA will fail)
Singular Value Decomposition
PCA can only be performed on squared covariance matrix, SVD can be applied directly to the input data matrix (more general than PCA)

$X = [x_1, x_2, \cdots x_n] \in \Re^{p\times n}$ is the matrix of mean-centered data of real values. SVD of X is of the form

$$X = U\Sigma V^T$$

where

- $U = [u_1, u_2, \cdots, u_l] \in \Re^{p\times l}$,
- $V = [v_1, v_2, \cdots, v_n] \in \Re^{l\times n}$,
- $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \cdots, \sigma_l) \in \Re^{l\times l}$ where $l = \min\{p, n\}$ and $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_l \geq 0$.

Properties:

- $\sigma_k, k = 1, 2, \cdots l$, are Singular Values of X.
- $Xv_k = \sigma_k u_k$. u_k are left Singular Vectors and are orthogonal, i.e., $U^TU = I$, where I is the identity matrix.
- $X^Tu_k = \sigma_k v_k$. v_k are right Singular Vectors and are orthogonal, i.e., $V^TV = I$, where I is the identity matrix.

PCA and SVD are both linear transforms/matrix factorization
Whenever we do PCA we can do SVD as well
SVD carries out 2 PCA's simultaneously one on the individual data points and one on the individual features

- The left singular vectors U in SVD are the same as the eigenvectors of the covariance matrix on features (C= DD^T)
- The right singular vectors V in SVD are the same as the eigenvectors of the covariance matrix on data points (D^{T}D)

Both the left and right singular vectors form orthonormal bases

Latent Semantic Analysis
An application of SVD for text document analysis (text mining)
Most important words/phrases for a set of documents
Words of similar meaning occur often in similar documents

- Step 1: Preprocessing D;
- Step 2: Construct a data matrix X from D;
- Step 3: Apply SVD to X;
- Step 4: Further analysis of SVD results depending on the application, e.g., document retrieval and document classification.
- Step 5: Interpretation of the results. This is application dependent and needs to be dealt with case by case.

Neural Networks

$$L(w) = \frac{1}{2}\sum_{d\in D}(t_d - y_d)^2,$$

Loss function that we will use for NN
Want to minimize the error using gradient descent

$$\begin{aligned}\nabla w_i = \frac{\partial L}{\partial w_i} &= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - y_d)^2\\ &= \frac{1}{2}\sum_{d\in D}2(t_d - y_d)\frac{\partial}{\partial w_i}\left(t_d - \sum w_i x_{id}\right)\\ &= \sum_{d\in D}(y_d - t_d)x_{id}\end{aligned}$$

General equation to reduce the error – following gradient descent
Can use SGD which instead randomly selects one data points and moves in the direction of the gradient on this data point instead of over all data points

$$w_i \leftarrow w_i - \eta\nabla w_i,$$

update to the weights in our weight vector

Stochastic Gradient Descent – select a random point from your dataset and backpropogate the error from that data point through your neural network vs
Gradient Descent – take all data points calculate loss (error) over all data points and backprop this error through the system
Backpropogation

- Initialize the network – start with random initial values of the model parameters (w)
- Push an input from the input to the output – compute the error for this example
- Back propogate the error at the output layer into network to compute the gradient

$$\nabla w'_{ij} = \frac{\partial L}{\partial w'_{ij}} = \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial u'_j}\frac{\partial u'_j}{\partial w'_{ij}}.$$

adjustments to weights at the hidden layer to output layer

From earlier,

$$\frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial u'_j} = (y_j - t_j)y_j(1-y_j); \qquad \frac{\partial u'_j}{\partial h_i} = w'_{ij}.$$

Since $h_i = f(u_i) = 1/(1+e^{-u_i})$ and $u_i = \sum_{k=1}^{p} w_{ki}x_k$,

$$\frac{\partial h_i}{\partial u_i} = h_i(1-h_i); \qquad \frac{\partial u_i}{\partial w_{ki}} = x_k.$$

Add up, we have

$$\nabla w_{ki} = \frac{\partial L}{\partial w_{ki}} = h_i(1-h_i)x_k\sum_{j=1}^{M}(y_j - t_j)y_j(1-y_j)w'_{ij}.$$

$$\nabla w_{ki} = \frac{\partial L}{\partial w_{ki}} = \sum_{j=1}^{M}\left(\frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial u'_j}\frac{\partial u'_j}{\partial h_i}\right)\frac{\partial h_i}{\partial u_i}\frac{\partial u_i}{\partial w_{ki}}.$$

adjustments to weights at input layer to hidden layer if using sigmod as activation

- Alternate among examples (SGD)

In general neural networks are *fully connected* meaning that each node is connected to all the nodes in the next layer
Deep Learning
A deep neural network is a network that has multiple layers, perceptron's with special functions or crazy network architecture

- Specialized programs tailored to specific problems

Autoencoder – special version of NN (trying to recreate inputs at the ouputs)

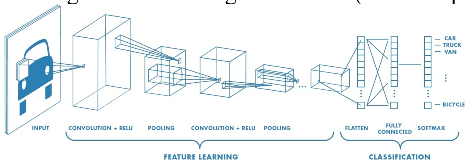
- Transform data to compressed, lower dimensional space then reconstruct the data from compressed version
- Has a symmetric structure (same number of input and output nodes)
- Use backprop to train autoencoder

Can be used for

- Data compression/pattern identification
- Pretraining a NN: find intrinsic features in the data that are potentially easier to find local minima through

Convolutional Neural Networks
Used for topographical data (data that is not the same if you swap the rows and columns)
Classification, localization (objects of interest in image), detection (localization of all objects in an image), segmentation (give a class label to every object in an image), outline (outline the contour of all objects of interest and label)
Convolution – way to combine matrices (measures similarity between two matrices sort of)
We have a sliding frame that we slide over the image to find different features/edge patterns in the image

Result of this operation is an activation map (shows where certain features can be identified in the image)
Initially the filter has random weights, at the end of training its weights capture features in the data
Pooling deals with image invariants (different positions)



$$f(x) = \max\{0, x\}$$

ReLU – non-linear activation function
Helps to solve the vanishing gradient problem (where the gradient gets so small it is not actually detecting the weights)
Vanishing gradient problem – as we back proprogate the gradient (error signal) decreases exponentially with n while the front layers train very slowly
NNMF

This is where NMF comes to help. The main idea of NMF rests on the "natural" parts that the original data (or observation) attend to capture and on the idea of representing or reconstructing the original data using such parts. In the math form, we have

- $X \approx WH$, where X is the data matrix, W is the matrix of parts (or bases or coordinates), and H is the matrix of weights of parts.
- To see the last point above, we focus on one data point x_i, $x_i \approx Wh_i$, where h_i is the weight of the parts in the data point x.

PCA Proof

- To understand why C may have orthonormal (i.e., orthogonal and of unit length) Eigenvectors, we have the following more general theorem.
- **Theorem 1**: A symmetric matrix is diagonalized by a matrix of its orthonormal eighenvectors.
- This means: $A \in \Re^{n\times n}$ with orthonormal Eigenvectors $E = [e_1, e_2, \cdots, e_n]$, then there exists a diagonal matrix Σ such that $A = E\Sigma E^T$.
- This is a strong claim with two parts.
- Part I: If e_i and e_j, $A = E\Sigma E^T$.
- **Proof**: In fact the proof doesn't require A to be symmetric but any matrix with Eigenvalues and Eigenvectors: (λ_k, e_k) for $k = 1, 2, \cdots, n$.
- By definition of Eigenvectors and Eigenvalues, $Ae_k = \lambda_k e_k$. Put that in the matrix form,
  $$A[e_1, e_2, \cdots, e_n] = [e_1, e_2, \cdots, e_n]\mathrm{diag}[\lambda_1, \lambda_2, \cdots, \lambda_n],$$
  which is $AE = E\Sigma$, where $E = [e_1, e_2, \cdots, e_n]$ and $\Sigma = \mathrm{diag}[\lambda_1, \lambda_2, \cdots, \lambda$
- Since E is orthonormal, i.e., $E^TE = EE^T = I$, where I is an identity matrix, this means that $E^T = E^{-1}$. In words, the inverse is the same as the transposition.

Now the real game - Derivation of SVD from PCA:

- Consider a given dataset $X \in \Re^{p\times n}$ again. We define $C_D = (n-1)C = X^TX \in \Re^{n\times n}$, which is the coverance matrix of data points, instead of features as we did in PCA, of X. C_D is symmetric.
- Based on Theorem 1, C_D has n pairs of Eighenvectors and Eighenvalues
  $$(v_1, \lambda_1), (v_2, \lambda_2), \cdots, (v_n, \lambda_n),$$
  so that $C_Dv_k = \lambda_k v_k$.
- Define $\sigma_k = \sqrt{\lambda_k}$, which are called Singular Values, and $u_k = \frac{1}{\sigma_k}Xv_i \in \Re^{p\times 1}$, or $Xv_k = \sigma_k u_k$.
- Now we claim:
  - $u_i^Tu_j = 1$ if $i = j$, or 0 otherwise, which means the two are orthonormal; and
  - $\|Xv_k\| = \sigma_k$.

  So $u_1, u_2, \cdots, u_n$ form an orthonormal basis of a Cartesian space. Formally, we have:
- **Theorem 2**: $Xv_1, Xv_2, \cdots, Xv_n$ form an orthonormal basis, where $\|Xv_k\| = \sigma_k$.
- **Proof**: To prove that Xv_i and Xv_j are orthogonal, we only need to show the dot product of the two is nonzero when $i = j$ or 0 when $i \neq j$.
- $(Xv_i)^T(Xv_j) = v_i^TX^TXv_j = v_i^TC_Dv_j = v_i^T\lambda_jv_j = \lambda_jv_i^Tv_j = \lambda_i$ if $i = j$, or 0 if $i \neq j$.

- So, $A = E\Sigma E^T$ follows.

- Part II: A is symmetric, then E is orthonormal.
- **Proof**: Consider (λ_i, e_i) and (λ_j, e_j) and $\lambda_i \neq \lambda_j$. Then
  $$\lambda_ie_i^Te_j = (\lambda_ie_i)^Te_j = (Ae_i)^Te_j = e_i^TA^Te_j = e_i^TAe_j = e_i^T\lambda_je_j = \lambda_je_i^Te_j.$$
- This means $\lambda_i = \lambda_j$, a contradiction which completes the proof.

- Now we are onto factorization:
- Let $V = [v_1, v_2, \cdots, v_n]$ and $U = [u_1, u_2, \cdots, u_n]$. Since $Xv_k = \sigma_ku_k$, $XV = [\sigma_1u_1, \sigma_2u_2, \cdots, \sigma_nu_n] = U\Sigma$, where $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \cdots, \sigma_n)$.
- This means that $X = U\Sigma V^{-1} = U\Sigma V^T$ since V is orthonormal so that $V^{-1} = V^T$.