k-Nearest Neighbours

$||x^{(a)} - x^{(b)}||_2 = \sqrt{\sum_{j=1}^{d}(x_j^{(a)} - x_j^{(b)})^2}$

Algorithm: 1. Find k examples (x^*, t^*) closest to **x**, that is $x* = \text{argmax}_{t^z} \sum_{r=1}^{k} \delta(t^{(z)}, t^{(r)}$ 2. Output $y = t*$

Choosing small k captures fine-grained patterns, may overfit. Choosing large k makes stable predictions by averaging, may underfit. $k < \sqrt{n}$ for n training examples

Curse of Dimensionality: In high dimensions, "most" points are far apart. Volume of a single ball for radius ϵ is $O(\epsilon^d)$ – we need $O((1/\epsilon)^d)$

Normalization: Nearest neighbors can be sensitive to ranges of different features. Fix by computing mean and s.d.

Variance: $Var(X) = E[(X-\mu)(X-\mu)^T]$

For constant Matrix, $Var(X + AY) = Var(X) + AVar(Y)A^T$

Covariance: For X, Y, $Cov(X,Y) = E[(X-\mu_X)(Y-\mu_Y)^T]$

Maximum Likelihood Estimator:
$\theta^{MLE} = \text{argmax} \prod_{i=1}^{N} p(X_i|\theta)$ To maximize MLE, take log of MLE so derivative calculation is easier.

Decision Trees

Accuracy gain: For decision tree with region R, split into R_1 and R_2 based on $L(R)$. $AG = L(R) - \frac{|R_1|L(R_1)+|R_2|L(R_2)}{|R_1|+|R_2|}$

Entropy: $H(X) = -E_{X\ p}[\log_2 p(X)] = -\sum p(x)\log_2 p(x) = -\sum\sum p(x,y)\log_2 p(x,y)$ High entropy: less predictable flat histogram, Low entropy: Histogram has peaks and valleys, predictable. For $H(Y|X=x), p(y|x) = p(x,y)/p(x)$ where $p(x)$ sum of row/column

Conditional Entropy: $H(Y|X) = \sum p(x)H(Y|X=x) = -\sum\sum p(x,y)\log_2 p(y|x)$ – H is always non-negative. **Chain rule**: $H(X,Y) = H(X|Y) + H(Y) = H(Y|X) + H(X)$. If X, Y and **indep**, $H(Y|X) = H(Y)$, but $H(Y|Y) = 0$. $H(Y|X) \leq H(Y)$

Information Gain: $IG(Y|X) = H(Y) - H(Y|X)$. If X is uninformative of Y, $IG = 0$. If X is completely informative of $Y, IG = H(Y)$

Decision Tree Algo: Split on most informative attribute, partitioning dataset, recurse on subpartitions. We desire small tree with informative nodes near the root.

Occam's Razor: Find simplest hypothesis that fits the observation

DT Problems: 1. Exponentially less data at lower levels. 2. Too big a tree can overfit data. 3. Mistakes at top-level propagate down tree.

DT Cont. attr.: Split based on a threshold, chosen to maximize IG – Decision tree can also be used for regression. Choose splits to minimize squared error, rather than maximize information gain.

Compared to kNN: Good with discrete attributes, missing values/Robust to scale of inputs/More interpretable. kNN handles features that interact in complex ways/incorp interesting distance measures (e.g. shape context)

Bias-Vraiance Decomposition

Mean squared error: $1/2(y-t)^2$

Training set: D, classifier: h_D, prediction: $h_D(x) = y$ where y is random var. D is random $\rightarrow h_D$ is random $\rightarrow h_D(x)$ is random

There is a distribution over the loss at x with $E_{D\ p_{dataset}}[L(h_D(x), t)]$.

We can now decompose the expected loss (suppress dist. x, D):

$E_{x,D}[h_D(x) - t^2] = E_{x,D}[h_D(x) - E_D[h_D(x)|x] + E_D[h_D(x)|x] - t)^2]$
$= E_{x,D}[(h_D(x) - E_D[h_d(x)|x])^2] + E_x[(E_D[h_D(x)|x] - t^2)^2]$ with $E_{x,D}$ being variance and E_x being the bias.

Bias: How close is classifier to true target (corresponds to underfitting)

Variance: How widely dispersed are our predictions as we generate new datasets? (corresponds to overfitting)

Bagging

Take a single dataset D with n examples. Generate m new datasets, each by sampling n training examples from D, with replacement. Average the predictions of the models train on each of these datasets.

Problem: Datasets are not indep. so there's no $1/m$ variance reduction. Possible to show that if var σ^2 and correlation ρ:
$Var(1/m \sum_{i=1}^{m} h_i(x)) = (1/m(1-p) + \rho)\sigma^2$

May be more advantageous to introduce more variability into algo, as long as it reduces the correlation between samples.

Bagging reduces overfitting by averaging predictions but does not reduce bias.

Random Forests: Bagged decision trees where when choosing each of node, choose split on random subset from d input features. Improves the variance reduction of bagging by reduced correlation b/w tree $\sim p$

Baye's Optimality: Hard to measure bias if t is not deterministic given x. Use measure distance from $y_*(x) = E[t|x]$

$E[(y-t)^2|x] = (y - y_*(x))^2 + Var[t|x]$

$$\mathbb{E}_{\mathbf{x},\mathcal{D},t|\mathbf{x}}[(h_{\mathcal{D}}(\mathbf{x}) - t)^2] =$$
$$\underbrace{\mathbb{E}_{\mathbf{x}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})] - y_*(\mathbf{x}))^2]}_{\text{bias}} + \underbrace{\mathbb{E}_{\mathbf{x},\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})])^2]}_{\text{variance}} + \underbrace{\mathbb{E}_{\mathbf{x}}[\text{Var}[t\,|\,\mathbf{x}]]}_{\text{Bayes}}$$

Bagging does not control Bayes error.

Regression

Loss function: Squared error $1/2(y-t)^2$

$y - t$ is the residual, want to make it small in magnitude.

Cost function: Loss function averaged over all training examples
$J(w,b) = 1/2\sum_i^N(y^{(i)} - t^{(i)})^2 = 1/2\sum_i^N(w^Tx^{(i)} + b - t^{(i)})^2$

Polynomial Regression: Fit data to M polynomial feature map: $y = w^T\psi(x)$ where $\psi(x) = [1, x, x^2, ...]^T$. Large M = large coefficients

L^2 **Regularization:** $R(w) = 1/2||w||_2^2$

Regularized cost function: $J_{reg}(w) = J(w) + \lambda R(w)$

Optimal weights are high, R is large. Poor data fitting results in large J. Makes tradeoff b/w fit to data and norm of weights. Large λ penalizes weight values more.

L^1 **regularization:** L^1 norm encourages weights to be exactly 0.

Linear methods for Classification

0-1 Loss: Hard to optimize due to step function. Changing weights by small amount has no effect on loss.

Linear Regression: Relaxation with smooth surrogate loss function, define loss as $w^Tx + b$. Loss function hates predictions with high confidence.

Logistic Activation Function: $\sigma(z) = 1/(1+e^{-z})$

Logit: $\sigma^{-1}(y) = \log(y/(1-y))$

Log-Linear: $L_{SE}(y,t) = 1/2 \quad (y-t)^2$ – Problem: $z << 0, \sigma(z) \approx 0$. If prediction is wrong, should be far from critical point.

Cross Entropy loss: $L_{CE}(y,t) = -t\log y - (1-t)\log(1-y)$
$\partial L_{CE}/\partial w_j = (y-t)x_j$

Logistic CE: $L_{LCE}(z,t) = L_{CE}(\sigma(z),t) = t\log(1+e^{-z}) + (1-t)\log(1+e^z)$ – Useful for when $y << 0$

Ridge Regression: when $\lambda \neq 0, w = (X^TX + \lambda I)^{-1}X^Tt$

Gradient Descent:

$w \leftarrow w - \alpha\frac{\partial J}{\partial w}$ α: learning rate

GD under L^2**:** $w \leftarrow (1-\alpha\lambda)w - \alpha\frac{\partial J}{\partial w}$ – Minimize regularized cost results in weight decay

Batch training: Taking gradient descent step over entire dataset of n examples (very slow)

Stochastic GD: Update parameters based on gradient of single training example,

1 – Choose i at random, 2 – $\theta \leftarrow \theta - \alpha[\partial L^{(i)}/\partial\theta]$

Problems: Variance of estimation can be high, not exploiting vectorized operations. Trick: Large learning rate at early stage of training, slowly decay over time. Makes loss appear to drop suddenly

Mini-batch: Compute gradient on medium-sized subset of examples. Note: SGD computed on larger mini-batch have smaller variance similar to bagging — Batched GD moves directly download, SGD is noisy but moves downhill on average.

Multiclass Classification

Softmax: $y_k = \text{softmax}(z_1, ..., z_K)_k = e^{z_k}/\sum_{k'} e^{z_{k'}}$ z_k are logits

Softmax CE: $L_{CE}(y,t) = -t^T(\log y)$

Neural Networks (MLP): $y = \phi(Wx + b)$

ReLU: $y = \max(0, z)$ **Soft ReLU:** $y = \log 1 + e^z$

Logistic: $y = 1/(1+e^{-z})$ **Hyperbolic:** $y = (e^z - e^{-z})/(e^z + e^{-z})$

Notes: Deep linear networks are no more expressive than linear regression. Multilayer feed-forward NN with nonlinear activation are universal function approximators. Logistic activation function are good for approximating hard threshold by scaling up weights and biases – they are also differentiable.

Univariate Chain Rule: $d(f(x(t)))/dt = df/dx * dx/dt$

Use $\bar{y}$ to denote derivative of dL/dy (aka error signal).

Back Prop.: Let $v_1, ..., v_n$ be topological ordering of computation graph, compute parents derivatives and propagate backwards to the previous nodes.

```
for i = 1..N
    Compute $v_i$ as function of $Pa(v_i)$
$\overline{v_N}$ = 1
for i = N-1..1
    $\overline{v_i} = \sum_{j \in Ch(v_i)} v_j \frac{\partial v_j}{\partial v_i}$
```

For $z = W_1x + b$, $h = \sigma(z)$, $y = W_2h + b_2$, $L = \frac{1}{2}||t-y||^2$,
$\overline{L} = 1$, $\overline{y} = \overline{L}(y-t)$, $\overline{W_2} = \overline{y}h^\top$, $\overline{b_2} = \overline{y}$, $\overline{h} = W_2^\top\overline{y}$, $\overline{z} = \overline{h} \circ \sigma'(z)$,
$\overline{W_1} = \overline{z}x^\top$, $\overline{b_1} = \overline{z}$.

Want $\frac{\partial\mathcal{L}}{\partial w}, \frac{\partial\mathcal{L}}{\partial b}$. In sigmoid activation with squared loss,

$$L = \frac{1}{2}(y-t)^2, y = \sigma(z), z = wx + b$$

$$\frac{\partial\mathcal{L}}{\partial y} = y - t, \frac{\partial\mathcal{L}}{\partial z} = \frac{\partial\mathcal{L}}{\partial y}\sigma'(z), \frac{\partial\mathcal{L}}{\partial w} = \frac{\partial\mathcal{L}}{\partial z}x, \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial z}$$

Derivative of Loss w.r.t. something is denoted with a bar. Get a topological ordering and compute the parents derivatives and propagate backwards to the previous nodes. In regularized, we have $\bar{w} = \bar{z}\frac{\partial \ddagger}{\partial w} + \bar{R}\frac{\partial\mathcal{R}}{\partial w}$. Computational cost forward is 1 add-multiply, backwards is 2.

numpy functions

```
np.norm
np.inv
np.dot
@ Matrix multiplication in numpy
np.transpose
np.identity
```

SVM

Hinge Loss: $\mathcal{L}_H(z,t) = \max(0, 1-zt)$

Support Vector Machines: If we use a linear classifier, with $z^{(i)}(\mathbf{w}, b) = \mathbf{w^T x} + b$, then we want to minimize hinge loss in training:

$$\min_{\mathbf{w},b}(\sum_{i=1}^{N}\max(0, 1 - t^{(i)}z^{(i)}(\mathbf{w}, b)))$$

Often used with L_2 regularization: $\min_{\mathbf{w},b}(\sum_{i=1}^{N}\max(0, 1 - t^{(i)}z^{(i)}(\mathbf{w}, b))) + \frac{\lambda}{2}||\mathbf{w}||_2^2$. Support vector machines maximizes the margin of the classifier. (maximizes the distance to the closest point in either class)

Boosting and AdaBoost

Boosting: training classifiers sequentially, focusing on training datapoints that were previously misclassified.

Weighted Training Sets: We can assign different weights (costs) for different examples. We have $w^{(n)}$ for every datapoint when we are calculating cost function. $\sum_{n=1}^{N} w(n)I[h(x^{(n)} \neq t^{(n)}]$

Weak Learner/Classifier: Informally, a classifier that is better than chance. The error of the classifier is at most $1/2 - \gamma$ for small $\gamma > 0$

Decision Stump: a decision tree with a single split.

Adaboost Algorithm:

1. Re-weight the training data by assigning larger weights to missclassified samples.
2. Train a new weak classifier with the new weighted data.
3. Add the new classifier to the ensemble. This ensemble is our new classifier.
4. Repeat until satisfied.

For $t = 1, \ldots, T$

- Fit a classifier to data using weighted samples ($h_t \leftarrow \text{WeakLearn}(\mathcal{D}_N, \mathbf{w})$), e.g.,

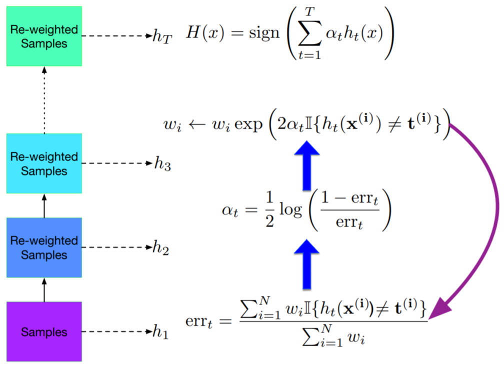
$$h_t \leftarrow \underset{h\in\mathcal{H}}{\text{argmin}} \sum_{n=1}^{N} w^{(n)}\mathbb{I}\{h(\mathbf{x}^{(n)}) \neq t^{(n)}\}$$

- Compute weighted error $\text{err}_t = \frac{\sum_{n=1}^{N} w^{(n)}\mathbb{I}\{h_t(\mathbf{x}^{(n)}) \neq t^{(n)}\}}{\sum_{n=1}^{N} w^{(n)}}$
- Compute classifier coefficient $\alpha_t = \frac{1}{2}\log\frac{1-\text{err}_t}{\text{err}_t}$ $(\in (0, \infty))$
- Update data weights

$$w^{(n)} \leftarrow w^{(n)}\exp\left(-\alpha_t t^{(n)}h_t(\mathbf{x}^{(n)})\right)\left[\equiv w^{(n)}\exp\left(2\alpha_t\mathbb{I}\{h_t(\mathbf{x}^{(n)}) \neq t^{(n)}\}\right)\right]$$

Return $H(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T}\alpha_t h_t(\mathbf{x})\right)$

Minimizes weighted error. Reduces bias by making each new classifier focus on previous mistakes. The training error of the output hypothesis is at most $L_N(H) = \frac{1}{N}\sum_{i=1}^N I\{H(x^{(i)} \neq t^{(i)} \leq exp(-2\gamma^2 T)\}$ Resilient to overfitting.


Re-weighted Samples
h_T
$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$
$w_i \leftarrow w_i \exp\left(2\alpha_t \mathbb{I}\{h_t(\mathbf{x}^{(\mathbf{i})}) \neq \mathbf{t}^{(\mathbf{i})}\}\right)$
Re-weighted Samples
h_3
$\alpha_t = \frac{1}{2}\log\left(\frac{1-\text{err}_t}{\text{err}_t}\right)$
Re-weighted Samples
h_2
Samples
h_1
$\text{err}_t = \frac{\sum_{i=1}^N w_i \mathbb{I}\{h_t(\mathbf{x}^{(\mathbf{i})}) \neq \mathbf{t}^{(\mathbf{i})}\}}{\sum_{i=1}^N w_i}$

Additive Models: A model with m terms is given by $H_m(x) = \sum_{i=1}^m \alpha_i h_i(x)$ – a linear combination of base classifiers $h_i(x)$ just like boosting
Stagewise training: 1. Initialize $H_0(x) = 0$ 2. For $m = 1..T$, compute m-th hypothesis $\alpha_m h_m$, and add it to the additive model $H_m = H_{m-1} + \alpha_m h_m$
Additive with Exponential Loss: TODO
Probablistic Models
Discriminative Approach: Estimate parameters of decision boundary/class separator directly from labeled examples **Generative Approach:** Model the distribution of inputs characteristic of the class
Baye's Classifier: Given features $x = [x_1, x_2, ..., x_D]^T$ we want to compute class probabilities using Bayes Rule $P(c|x) = \frac{p(x|c)}{p(x)} = \frac{p(x|c)p(c)}{p(x)}$, formally, posterior $= \frac{\text{class likelihood} \times \text{prior}}{\text{evidence}}$
Log Likelihood: $\sum_{i=1}^N \log p(c^{(i)}) = \sum_{i=1}^N c^{(i)} \log \pi + \sum_{i=1}^N (1 - c^{(i)}) \log(1-\pi)$
Naive Bayes (Inference): Apply Bayes' Rule, shorthand notation: $p(c|x) \propto p(c)\prod_{j=1}^D p(x_j|c)$ – cheap learning algorithm. At training time, estimate parameters using log likelihood, compute co-occurrence counts of each feature with the labels. At test time, apply Bayes' Rule.
MLE Want to maximize the log likelihood: $l(\theta) = \sum log(p(X_i|\theta))$. Too little data can cause overfitting.
Bayesian Parameter Estimation: MLE treat observation as random variables, but parameters are not. Bayesian treats parameters as random variables as well. Define *prior distribution* $p(\theta)$, which encodes beliefs about parameter before observing data. *likelihood* $p(D|\theta)$, same as MLE. Updating beliefs by computing posterior distribution using Bayes' Rule:
$$p(\theta|D) = \frac{p(\theta)p(D|\theta)}{\int p(\theta')p(D|\theta')d\theta'}$$
MAP Estimator: Find the most likely parameter settings under the posterior Want to maximize the log likelihood: $l(\theta) = \log(p(w)) + \sum log(p(X_i|\theta))$.
$\theta_{MAP} = \text{argmax}_\theta \log p(\theta) + \log p(D|\theta)$
GMM: $\Sigma = Cov(x^{(i)}) = E[(x^{(i)} - \mu)(x^{(i)} - \mu)^T$
Principle Componenet Analysis (PCA): Reduces the dimension of data which can help with saving computation/memory and reducing overfitting.
Given datapoints, consider the sample mean $\hat{\mu}$ and the sample covariance $\hat{\Sigma} = \frac{1}{N}\sum(x^{(i)} - \hat{\mu})(x^{(i)} - \hat{\mu})^\top$. Can find a subspace S of $\mathbb{R}^D$ with orthonormal basis $u_1, ..., u_K$ that "best represents" our $x^{(i)} - \hat{\mu}$ (or minimizes reconstruction squared error loss or maximizes variance of reconstructions (sum of squares of distances to the mean)) by doing spectral decomposition of $\hat{\Sigma}$, sorting the eigenvalues (which are all non-negative as $\hat{\Sigma}$ is positive semidefinite) from biggest to smallest, then taking the first K eigenvectors as $u_1, ..., u_k$. We project points $x^{(i)} - \hat{\mu}$ onto S by applying the matrix $U^\top$ for $U = (u_1, ..., u_k)$. We can then "unproject" by applying U. The features of the projected datapoints are uncorrelated.
Deriving PCA: Maximize $a^T \Lambda a = \sum_{j=1}^D \lambda_j a_j^2$ for $a = Q^T u$, assume $_i$ are in descending order, by observation, since u is a unit vector, then by unitarity, a is also a unit vector: $a^T a = u^T Q Q^T u = u^T u$. By inspection, set $a = \pm 1$, and $a_j = 0$ for $j \neq 1$. Hence, $u = Qa = q_1$ (the top eigenvector)
Autoencoder: An autoencoder is a feed-forward neural net whose job is to take an input x and predict x. Obviously trivial until we mandate that one of the layers has to have fewer features than the input.
If it's got linear activation functions and SE loss, then clearly we should just do PCA.
Otherwise we have that it projects data onto a manifold (image of the autoencoder) instead of a subspace. This is like a nonlinear dimensionality reduction.
K-means: Group data points into clusters. Relies on assumption that data depends on latent variables. The objective:
$\min_{\{\mathbf{m}_k\},\{\mathbf{r}^n\}} \sum_{n=1}^N \sum_{k=1}^K r_k^{(n)} ||\mathbf{m}_k - \mathbf{x}^{(n)}||^2$
Where $\{\mathbf{m}_k\}$ are the k means, $\mathbf{r}^n \in \mathbb{R}^K$ is the assignment vector for $\mathbf{x}^n$. $\mathbf{r}^n$ is one-hot if hard k-means and like a probability distribution if soft k-means. Finding the min is NP-hard in general.
Alternating minimization involves fixing one of $\{\mathbf{m}_k\}$, or $\{\mathbf{r}^n\}$, optimizing on that, then alternate to other. First initialize clusters, then iterate between assignment step (assign data points to nearest clusters), and refitting step (move every cluster centre to the means of the data points assigned to it). This algo is guaranteed to converge to a local optimum. **Test of convergence**: if the assignments don't change over an iteration, then stop. **Refitting Step**: For each centre: $\mathbf{m}_k = (\sum_n r_k^{(n)} \mathbf{x}^n)/(\sum_n \mathbf{r}_k^{(n)})$.
Assignment Step: For soft k-means:
$$r_k^{(n)} = (\exp(-\beta||\mathbf{m}_k - \mathbf{x}^n||^2))/(\textstyle\sum_j \exp(-\beta||\mathbf{m}_j - \mathbf{x}^n||^2)))$$
$$\mathbf{r}^{(n)} = \text{softmax}(-\beta\{||\mathbf{m}_k - \mathbf{x}^n||^2\}_{k=1}^K)$$
For regular: $r_k^{(n)} = \mathbb{I}[k'^{(n)} = k]$, for $k = 1, ..., K$, where $k'^{(n)} = \text{argmin}_k ||\mathbf{m}_k - \mathbf{x}^{(n)}||^2$. As β goes to infinity, soft k-means becomes regular.
Gaussian Mixture Models: A generative approach to clustering. We need a method to judge what it means to cluster the data well. Assume that the data was generated from a model. We then adjust the model parameters to maximize the probability that the data was generated by our model. Given $\mathcal{D}$, we assume that the data point, $\mathbf{x}$, was generated by choosing a cluster $k \in \{1, ..., K\}$, such that $p(z = k) = \pi_k$, and sampling from $\mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$, i.e. $p(\mathbf{x}|z = k) = \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$. To find the best parameters, we want to max:
$$\log p(\mathcal{D}) = \sum_{n=1}^N \log p(\mathbf{x}^{(n)}) = \sum_{n=1}^N \log\left(\sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \mathbf{\Sigma}_k)\right)$$
We can find MLE using gradient descent, or alternating optimization like k-means. Supposedly a universal approximator. (whatever that means)
EM Algorithm for GMM: A local algorithm for GMM. First initialize the starting μ_k and π_k. Then iterate between the E step and M step until convergence (check by looking at log likelihood).
E step: posterior probability over z over current model (how much we think a datapoint was generated by a cluster):
$$r_k^{(n)} = p(z^{(n)} = k|\mathbf{x}^{(n)}) = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(\mathbf{x}^{(n)}|\mu_j, \mathbf{\Sigma}_j)}$$
M step: update the model parameters π_k, μ_k to optimize the expected complete data log-likelihood. $\mu_k = (1/N_k)\sum_{n=1}^N r_k^{(n)} \mathbf{x}^{(n)}$, $\pi_k = N_k/N$, where $N_k = \sum_{n=1}^N r_k^{(n)}$.
Matrix Completion: For an $N \times M$ sparse matrix R (think users by movies for Netflix), we want to factor it into $N \times K$ matrix U and a $M \times K$ matrix Z, for $K << N, M$. In other words we want to minimize $||R - UZ^\top||$ with norm being the sum of the squares of the entries. This is an NP-Hard non-convex problem, but it's easy if one of U and Z is fixed. So initialize U and Z randomly, then for $O = \{(m, n)$: that's one of the filled in entries$\}$, alternate doing $u_n = \left(\sum_{m:(n,m)\in O} |z|^2\right)^{-1} \sum_{m:(n,m)\in O} R_{nm} z_m$ and
$$z_m = \left(\sum_{n:(n,m)\in O} |u_n|^2\right)^{-1} \sum_{m:(n,m)\in O} R_{nm} u_n.$$
Alternatively, can do stochastic gradient descent, and randomly pick a $(n, m) \in O$, then set $u_n = u_n - \alpha(R_{nm} - u_n^\top z_m)z_m$ and $z_m = z_m - \alpha(R_{nm} - u_n^\top z_m)u_n$.
Matrix Factorization: PCA has $x^{(i)} \approx Uz^{(i)}$, so $X \approx ZU^T$, where X, Z have rows of Data points. Frobenius Norm is the L2 distance between all points of 2 matrices, so $X - ZU^T{}_F^2 = X^T - UZ^T{}_F^2$, so we approximate a DxN (X^T) matrix with DxK (U) matmul KxN(Z^T) where U is chosen to minimize reconstruction error and is the optimal rank K approximation. This is similar to SVD where $X = QSU^T$ where Q, U is orthonormal and S is a diagonal with nonnegative terms ordered from largest to smallest. Consider positive semidefinite XX^T (NxN) and $X^T X$ (DxD). $XX^T = QSU^T USQ^T = QS^2 Q^T$ is an eigendecomposition of XX^T. Similarly, $X^T X = US^2 U^T$ is a eigendecomposition of $X^T X$. We can show that these share D eigenvalues and the rest are zeroes. The optimal PCA subspace is spanned by the top K eigenvectors of $\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^N (x^{(i)} - \hat{\mu})(x^{(i)} - \hat{\mu})^T$, whe the data is centered, we have $\hat{\Sigma} = \frac{1}{N} X^T X = U\Lambda U^T$, the SVD of $X = QSU^T = ZU^T$, so $\hat{\Sigma} = \frac{1}{N} USQ^T QSU^T = U\frac{S^2}{N}U^T$, so the eigenvalues are the singular values $\lambda_i = \frac{s_i^2}{N}$. Since A is semi definite positive, so eigenvectors are perpendicular and $Q^{-1} = Q^T$
Recommendation: Use sparse rating matrix R to find latent factors to guess at missing data. Assume rating of user n on movie m is given by $R_{nm} = u_n^T z_m$, so matrix form is UZ^T with users as rows and movies as columns. This is a matrix factorization where we want to minimize $\sum_{i,j}(R_{ij} - u_i^T z_j)^2$, since most of it is 0, we can consider the observed set O and calculate squared error loss: $\frac{1}{2}\sum_{n,m\in O}(R_{nm} - u_n^T z_m)^2$. Solve this with Alternating Least Squares on Z and U.
$$u_n = \left(\sum_{m\in O} z_m z_m^T\right)^{-1} \sum_{m\in O} R_{nm} z_m$$
$$z_m = \left(\sum_{n\in O} u_n u_n^T\right)^{-1} \sum_{n\in O} R_{nm} u_n$$
Can also use Gradient Descent, but it is expensive if all indices are considers and R is large. SGD is better, for a randomly chosen pair, update:
$$u_n - = \alpha(R_n m - u_n^T z_m)z_m$$
$$z_m - = \alpha(R_n m - u_n^T z_m)u_n$$
K-means can also also be thought of as $X \approx RM$, ordering the R leads to a vertical block shape, then co-clustering clusters both rows and columns of R, giving it a blocky shape.
Reinforcement Learning: We observe inputs, and have to choose correct output (actions) in order to maximize rewards. Correct outputs not provided. Challenges in RL include: dealing with a continuous stream of data, the effect of action depends on state on world, we only know the reward for an action, and there could be a delay between action and reward. **Markov Decision Process** (MDP) the framework we use to describe RL problems. A **discounted MPD** is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is state space, $\mathcal{A}$ is finite action space, $\mathcal{P}$ is transition probability, $\mathcal{R}$ is immediate reward, and γ is discount factor $0 \leq \gamma \leq 1$.
State-value function: describes the expected discount reward if the agent starts from state s and follows policy π