CS M146 Notes

Nikola Samardzic

March 19, 2018

1 Key learning concerns

- Key modeling questions:
  - What are the inputs/outputs?
  - What is the hypothesis space?
  - What is the loss function?
- Key algorithm questions:
  - How are predictions made?
  - How to learn the model?
- Key analysis questions:
  - What are the properties/guarantees of the model?
  - What are the pros/cons/connections to other models?

2 Data

- *Training data*: Used to learn model parameters
- *Test data*: Used to asses how a trained model will do in predicting unseen data.
- *Development data*: Used to optimize hyperparameters
- *N-fold cross validation*:

  - Split data into N equal-sized parts
  - Train and test N different classifiers
  - Retrain the classifier with best cross validation performance, and retest.

3 Decision Trees

- Algorithm: Learning decision trees (ID3)

  (a) Split data based on feature that gives *highest information gain*

  (b) If a random variable S has K different values, a_1, a_2, ..., a_K, its entropy is given by:

  $$H[S] = -\sum_{v=1}^{K} \Pr(S = a_v) \log \Pr(S = a_v)$$

  and information gain is defined as:

  $$\text{Gain}(S, A) = H[S] - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H[S_v]$$

- Classification: Start from the root and walk down the tree until you reach leaf node
- Advantages: Compactly presents a lot of data
- Disadvantages: Easy to overfit with decision trees

4 K-Nearest Neighbors (KNN)

- Classification: For discrete values, predict the most frequent label among the neighbors. For continuous values, predict the average of KNN.
- L_p-norm:

  $$||x_1 - x_2||_p = \left(\sum_{i=1}^{n} |x_{1,i} - x_{2,i}|^p \right)^{1/p}$$

- Hyperparameters: Number of neighbors (k), and distance metric.

- Neighbor's labels could be weighted by their distance.
- Issue of feature normalization: You probably want to center all data at zero mean, unit standard deviation.
- KNN is type of *instance-based learning* (classify based on similar stored data)
- Advantages: Training is fast; can learn very complex functions; training data is stored
- Disadvantages: Needs lots of storage; prediction can be slow; can be fooled by irrelevant attributes; curse of dimensionality (things go wrong in high dimensions). Both irrelevant features and curse of dimensionality can be addressed via dimensionality reduction and feature selection.

5 Linear Classification and Perceptron

- Classification via *Linear Threshold Units (LTU)*: $\text{sgm}(w^T x + b)$.
- Many other linear classifiers: Support Vector Machines, Logistic Regression
- Guranteed to work for linearly separable data
- Updates on mistaken prediction (x, y): $w_{t+1} = w_t + x$.
- Margin of given w vs. margin of the dataset
- **Thoerem:** Assume training set $\{(x_1, y_1), ..., (x_m, y_m)\}$ and for all i, the feature vector $x_i \in \mathbb{R}^n$, $||x_i|| \leq R \in \mathbb{R}$ and $y_i \in -1, +1$. Suppose there exists a unit vector $u \in \mathbb{R}^n$ such that for some $\gamma > 0$ we have $y_i(u^T x_i) \geq \gamma$ for all i (i.e., the margin of the dataset is γ). Then, the perceptron will make at most R^2/γ^2 mistakes on the training data.

  Proof: Let's look at what happens after we made t mistakes.

  - Claim 1: $u^T w_t \geq t\gamma$. Initialize $w_0 = 0$. Then, for some j, $u^T w_1 = u^T x_j \geq \gamma$. We know that $w_t = w_{t-1} + x_i$, so $u^T w_t = u^T w_{t-1} + u^T x_i \geq (t-1)\gamma + \gamma = t\gamma$.

  - Claim 2: $||w_t||^2 \leq tR^2$. Initialize $w_0 = 0$. Then, for some j, $||w_1||^2 = w_1^T w_1 = x_j^T x_j \leq R^2$. We know that $w_t = w_{t-1} + x_i$, so

$$\begin{aligned} ||w_t||^2 &= w_t^T w_t = (w_{t-1} + x_i)^T (w_{t-1} + x_i) \\ &= ||w_{t-1}||^2 + ||x_i||^2 + 2w_{t-1}x_i \\ &\leq (t-1) + R^2 = tR^2 \end{aligned}$$

    because $||x_i||^2 \leq R^2$ and $w_{t-1}x_i < 0$ because it is a mistake.
  - Claim 3: $R\sqrt{t} \geq ||w_t|| \geq u^T w_t$. By Cauchy-Schwartz, $1 \cdot ||w_t|| = ||u|| \cdot ||w_t|| \geq u^T w_t \geq t\gamma$.

  Thus, $R\sqrt{t} \geq u^T w_t \geq t\gamma$, i.e.

$$\boxed{t \leq \frac{R^2}{\gamma^2}.}$$

- Advantages: Perceptron makes no assumption about data distribution (online learning alogrithm), the distribution can even be adversarial; converges after fixed number of mistakes.
- Disadvantages: Does not work on data that is not linearly separable.
- Perceptron with finite data set: Learn done through multiple *epochs* (a new hyperparameter) that shuffle data around on each turn. Need learning rate (another hyperparameter) for each epoch.
- Voting and averaging: Instead of just submitting final weight vector, aggregate parameters on learning path. Especially good when the data is not separable.
  - *Voted*: Remember every weight along the path. The most survived weight wins.
  - *Averaged*: Use average weight vector. More practical and popular.
- *Batch learning*: All training data used at training time; training usually done by error minimization. Assumes independent, identically distributed training sample.
- *Online learning*: Update the model whenever new data comes. Good when data distribution changes over time, when data is streamed, or when dataset is huge. Does not assume independent, identically distributed data.

6 Logistic Regression

- Great when data is not linearly separable
- Goal is to learn a linear classifier, but instead of predicting output, predit $\Pr(y = 1|x)$.
- Training through regression
- Model assumes $\Pr(y = 1|x) = \sigma(w^T x + b)$, where

$$\sigma(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}.$$

- Hypothesis space: $h_w(x) = \sigma(w^T x)$
- Classification: If $\sigma(w^T x) > 1/2$, predict 1.
- Likelihood: The likelihood function, given training data $\{(x_i, y_i)\}$ given by

$$L(\theta) = \Pr(y_1, y_2, ..., y_n|x_1, x_2, ..., x_n; w)$$

  Assuming training data is i.i.d.:

$$L(\theta) = \prod_i \Pr(y_i|x_i; \theta)$$

- Learning: Maximize likelihood on w using gradient descent (no closed form solution, but function is convex):

$$\begin{aligned} \underset{w}{\operatorname{argmax}}\, L(\theta) &= \underset{w}{\operatorname{argmax}} \prod_i P(y_i|x_i; w) \\ &\boxed{= \underset{w}{\operatorname{argmin}} \sum_i \log(1 + \exp(-y_i w^T x_i))} \end{aligned}$$

- Update rule for *gradient descent*: $\theta^{(t+1)} = \theta^{(t)} - \eta \nabla J\left(\theta^{(t)}\right)$, where η is the *learning rate.*

7 Proving Convexity

To prove $f(\mathbf{x})$ is convex, where $\mathbf{x} \in \mathbb{R}^n$, prove the Hessian is positive semi-definite:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

that is, prove that for all z, $z^T H z = \sum_{j,k} H_{j,k} z_j z_k \geq 0$.

Also can use *Jensen's inequality*, prove

$$f(\lambda a + (1-\lambda)b) \leq \lambda f(a) + (1-\lambda) f(b)$$

for all a, b, and $0 \leq \lambda \leq 1$.

8 Linear Regression

- Predicts a continuous outcome variable.
- Loss function: Least Mean Squares loss

$$J(w) = \frac{1}{2} \sum_{i=1}^{m} (y_i - w^T x_i)^2$$

- Learning algorithm: Least Mean Squares regression: Until convergence update $w = w - \eta \nabla J(w)$.
  - How to find $\nabla J(w)$? First,

$$\nabla J(w) = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, ..., \frac{\partial J}{\partial w_d} \right]$$

    For least mean squares, we have

$$\frac{\partial J}{\partial w_i} = \frac{\partial}{\partial w_i} \sum_{j=1}^{m} (y_j - w^T x_j)^2 = -\sum_{j=1}^{m} (y_j - w^T x_j) x_{j,i}$$

  - Since the above computation is slow, we can use *Stochastic/Gradient Descent*: Until convergence randomly pick an example, pretend that the entire training set is represented by this single example, use this example to calculate the gradient and update the model. This way, if we are processing the j^{th} example, we have

$$\frac{\partial J}{\partial w_i} = (w^T x_j - y_j) x_{j,i}$$

9 Computational Learning Theory

Two general models of learning:

(a) Analyze the probabilistic intuition: Never saw a feature in the positive examples? Maybe we will never see it; and if we do, it will be with small probability, so the concepts we learn are still pretty good in terms of performance on future data. PAC framework.

(b) Mistake-driven learning: Update hypothesis only when you make a mistake. Define good in terms of how many mistakes you make before you stop; e.g. online learning; no answer to "How many examples do you need before convergence?"; no assumptions on order or distribution of sample data.

9.1 PAC learning (Probably Approximately Correct)

- A model for *batch learning*: training done on fixed dataset and then deployed.
- Assumes fixed data distribution
- *Error of hypothesis*: Given a distribution D over examples, the error of hypothesis h with respsect to target concept f is $\mathrm{err}_D(h) = \Pr_{x \sim D}[h(x) \neq f(x)]$
- *Empirical error*: Given training set S, $\mathrm{err}_S(h) = \Pr_{x \in S}[h(x) \neq f(x)]$
- A *boolean domain* is any domain of the type B^k, $k \in \mathbb{N}$, $B = \{0, 1\}$.
- A *boolean function over a boolean domain* X is any function of the type $f : X \to \{0, 1\}$.
- A *concept* over a domain X is a total Boolean function over X.
- A *concept class over a domain* is a set of concepts over that domain.
- *PAC learnability*: The concept class C over a domain X is PAC learnable by a learner L using a hypothesis space H if for all $f \in C$, and all distributions D over X, and fixed $\epsilon > 0$, $\delta < 1$, given m examples sampled i.i.d. to D, the learner L produces, with probability at least $1 - \delta$, a hypothesis $h \in H$ that has error at most ϵ, where m is polynomial in $1/\epsilon$, $1/\delta$, n, and $\mathrm{size}(H)$. That is, for a given m we have

$$m > \frac{1}{\epsilon}\left[\log \mathrm{size}(H) + \log \frac{1}{\delta}\right]$$

- What is size(H) is inifite?
- A set S of examples is *shattered* by a set of functions H if for every partition of the examples in S into positive and negative examples, there exists a function in H that assigns exactly these labels.
- The *VC Dimension* of a hypothesis space H over instance space X is the size of the largest finite subset of X that is shattered by H.

10 Kernels

- Use linear models to fit non-linear data by changing the data.
- *Dual representation*: The weight vector w can be written as a linear combination of examples $w = \sum \alpha_i y_i x_i$, where α_i is the number of mistakes on the i^{th} example. So our prediction can be written as $w^T x = \sum_i \alpha_i y_i x_i^T x$. With kernels this becomes $\sum_i \alpha_i y_i \phi(x_i)^T \phi(x) = \sum_i \alpha_i y_i K(x_i, x)$
- *Polynomial kernel*: $K(x, z) = (1 + x^T z)^n$.
- *Gaussian kernel*: $K(x, z) = \exp\left[-\frac{||x-z||^2}{c}\right]$.
- Kernels must be inner products in some feature space. How to check if something is a kernel? Construct *Gram matrix* $\{K(x_i, z_j)\}$ and check that it is positive semi-definite, i.e. it is symmetric and for any vector z, $z^T\{K(x_i, z_j)\}z \geq 0$. Also, can use *Mercer's condition*: for every finite set of vectors $\{x_1, x_2, ...\} \subset \mathbb{R}^n$, and any choice of $\mathbf{c} \in \mathbb{R}^n$, $\sum_i \sum_j c_i c_j K(x_i, x_j) \geq 0$ must hold.

11 Support Vector Machines

- *Marginal perceptron algorithm*: Adds margin hyperparameter γ. Run perceptron to train, but make update if $y_i(w^T x_i) \leq \gamma$ (instead of running only when $y_i(w^T x_i) < 0$.
- *Theorem (Vapnik)*: $\text{VC}(H) \leq \min\left(R^2/\gamma^2, d\right) + 1$
- *Geometric margin*: $\gamma = \min_{x_i, y_i} \frac{y_i(w^T x_i + b)}{||w||}$.
- For practical reasons, fix $\min_{x_i, y_i} y_i(w^T x_i + b) = 1$, then $\gamma = ||w||^{-1}$.

- This is now called the *hard support vector machine.*
- *Soft SVM*: Allow some examples to break into the margin or even make a mistake.
- Introduce one *slack variable* ξ_i per example and require that $y_i(w^T x_i + b) \geq 1 - \xi_i$, $\xi_i \geq 0$.
- Loss function: We want maximum margin ($w^T w$) and minimal loss ($\sum_i \xi_i$):

  $$\min_{w,b,\xi_i} \frac{1}{2} w^T w + C \sum_i \xi_i$$

  where C is a hyperparameter that specifies the tradeoff between the loss and margin. Equivalently:

  $$\min_{w,b,\xi_i} \frac{1}{2} w^T w + C \sum_i \max(0, 1 - y_i(w^T x_i + b))$$

  This is called *hinge loss.*

  - i^{th} example is incorrectly classified if $\xi_i > 1$
  - i^{th} example is within the margin, but correctly classified if $0 > \xi_i > 1$
  - i^{th} example is correctly classified and outside margin if $\xi_i = 0$
- The loss function is convex but not differentiable everywhere, so introduce *subgradient*:

  $$\nabla J^t = \begin{cases} w & \text{if } \max(0, 1 - y_i(w^T x_i + b)) = 0 \text{ (i.e., outside margin)} \\ w - C y_i x_i & \text{otherwise} \end{cases}$$

- The Soft SVM Perceptron:

  $$w^{(t)} = \begin{cases} w - \eta w^{(t-1)} & \text{if } y(w^{(t-1)T} x + b) \geq 1 \\ \eta C y x & \text{otherwise} \end{cases}$$

- *Dual form representation*: The final w can be represented as $w = \sum_i \alpha_i y_i x_i$ where if $\alpha_i = 0$ then $y_i(w^T x_i + b) > 1$ (i.e., point correctly classified, outside of margin), if $\alpha_i = C$ then $y_i(w^T x_i + b) < 1$ (i.e., point incorrectly classified) and $0 \leq \alpha_i \leq C$ then $y_i(w^T x_i + b) = 1$ (i.e., point is on the margin). The x_i's for which $\alpha_i \neq 0$ are called support vectors.

12 Ensamble Methods

- Yet another way to introduce non-linearity in linear models; Combines models for better results.
- *Bagging* (Bootstrapping Aggregation): Train multiple models; final calssifier takes votes from each model. If changing the training set can cause significant changes in the learned classifier then bagging can improve accuracy.
- *Boosting Algorithm*:
  (a) Given training set $\{(x_1, y_1), ..., (x_m, y_m)\}$
  (b) For epochs $t = 1, 2, ..., T$:
  (c) Construct a distribution D_t on $\{1, 2, ..., m\}$
  (d) Find a weak hypothesis h_t such that it has a small weighted error ϵ_t on D_t.
  (e) Construct final output H_{final}.
- *AdaBoost*: One approach to Boosting
  - Initialize $D_1(i) = \frac{1}{m}$ for all i.
  - Find best hypothesis given the current weights
  - Change weight given following rule:

    $$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \begin{cases} e^{-\alpha_t} & \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if } y_i \neq h_t(x_i) \end{cases}$$

    where Z_t is a normalization constant that ensures that $\sum_i D_{t+1}(i) = 1$, and $\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$, where ϵ_t is the weighted error of h_t on the training data.
  - Construct $H_final = \text{sgn}\left[\sum_t \alpha_t h_t(x)\right]$.
- *Theorem*: Running AdaBoost for T rounds, and assuming $\epsilon_t = \frac{1}{2} - \gamma_t$, $\gamma_t > 0$ (i.e., each hypothesis is better than random) and let $\gamma = \min_t \gamma_t$. Then the training error is $\leq e^{-2\gamma^2 T}$. That is, the training error of the combined classifier decreases exponentially fast as long as the errors of the weak classifiers are strictly better than chance.
- Bagging and Decision Trees combined give *random forest*, which reduces gradient of resulting classifier.

13 Multiclass Models

- Used to model data that fits into more than 2 classes (but each datapoint is still only assigned one class).
- Two approaches: Reduce multiclass problem to binary classification, or train a single multiclass classifier.
- Pros of reducing to binary classification: If binary classification improves in future, so odes multiclass reduction; easy to implement
- Pros of using single multiclass classifier: Global optimization (i.e., loss can be minimized directly); easy to add constraints and domain knowledge.
- *One-against-all learning*: Make K classifieres that separate one class from all others; winner takes all $y = \text{argmax}\left(w_{\mathcal{C}_1}^T x, w_{\mathcal{C}_2}^T x, ..., w_{\mathcal{C}_K}^T x\right)$; assumes each class linearly separable from all others.
- *One vs. one learning*: Making decision is more comlpex: each label gets $k-1$ votes; outputs of different classifiers may not be coherent; Majority vs. Tournament voting; Assumes every pair of classes is linearly separable; needs a lot of memory to store; possibility of overfitting
- *Perceptron style multiclass classification*: For each example (x, y), for each label $y' \neq y$, if $w_y^T x < w_{y'}^T x$ (i.e., if the model produces mistaken prediction), then promote y by $w_y = w_y + \eta x$, and demote y' by $w_{y'} = w_{y'} - \eta x$.

14 Empirical Risk Minimization

- We will study methods of directly solving multiclass problems (without reducing to binary classification)
- Linear classification methods we learned so far:
  - SVM:

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_i \max(0, 1 - y_i(w^T x_i + b))$$

  - Logistic Regression:

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_i \log(1 + log(-y_i(w^T x_i + b))$$

  - Linear Regression:

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_i (y_i - (w^T x_i + b))^2$$

- *Regularized loss minimization*: We want to learn

$$\min_{h \in H} R(h) + \frac{1}{m} \sum_i L(h(x_i), f(x_i))$$

  where f is the objective function, and h is the hypothesis, R is the regularization term used to prevent overfitting, and L is the loss function.

- *Multiclass loss function*: Many options

  -
$$L_{0-1}(y, y') = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{if } y = y' \end{cases}$$

    equivalently

$$L_{0-1}(y, x, w) = \begin{cases} 1 & \text{if } y w^T x \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

    These are ideal models, but cannot be used because not differentiable.

  - Hinge loss (SVM):

$$L(y, x, w) = \max(0, 1 - y w^T x)$$

  - Perceptron:

$$L(y, x, w) = \max(0, -y w^T x)$$

  - Exponential (AdaBoost):

$$L(y, x, w) = \exp(-y w^T x)$$

  - Logistic regression:

$$L(y, x, w) = \log(1 + \exp[-y_i(w^T x + b)])$$

- Regularization can be sparse ($L - 0$, $L - 1$) or Gaussian prior ($L - 2$)

15 Multiclass Approaches

- *Clustering*: Given collection of data points, the goal is to find structure in the data without any labels; can use any norm for distance
- *K-means algorithm* (Lloyd's algorithm): Greedy algorithm for minimizing K-means objective: randomly assign the cluster center $\{\mu_k\}$; then minimize loss over the points by assigning every point to the closes cluster center; then minimize loss by updating cluster center to center of their current cluster; and loop until convergence. Converges to local minimum in exponential number of training points.
- *K-medoids algorithm*: Same as K-means, except that the centers are assigned the example point closest to the cluster mean instead of the mean itself.
- *Gaussian mixture models*: Model each cluster as fitting to a distinct Gaussian distribution. Parameters that need to be trained are the (μ, Σ) pair for each cluster:

  $$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right]$$

  and given training cluster $\{x_1, x_2, ..., x_m\}$:

  $$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \Sigma = \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu)(x_i-\mu)^T$$

  Also assign weights to each of the clusters based on number of sample points in each:

  $$w_k = \frac{\sum_n \gamma_{nk}}{\sum_k \sum_n \gamma_{nk}}$$

  where $\gamma_{nk} = 1$ if example point n belongs to cluster k, and 0 otherwise. For a given test point x, make decision based on the highest Gaussian pdf value out of all the clusters:

  $$\begin{aligned}\gamma_{nk} &= \Pr(Z_n = k | x_n) \\ &= \frac{\Pr(x_n | Z_n = k)\Pr(Z_n = k)}{\Pr(x_n)} \\ &= \frac{\Pr(x_n | Z_n = k)\Pr(Z_n = k)}{\sum_{k'} \Pr(x_n | Z_n = k')\Pr(Z_n = k')}\end{aligned}$$

Also, the prior for any point x is given by:

$$\Pr(x) = \sum_{k=1}^{K} w_k \text{pdf}(x|\mathcal{N}(\mu_k, \Sigma_k))$$

- What when cluster assignment are not given? Use Expectation Maximization: initialize the Gaussians randomly (call all the Gaussian parameters θ); then compute the γ_{nk}'s given θ; then update θ using the γ_{nk}'s; repeat until convergence)

16 Bayesian Learning

- Goal: Find the best hypothesis from some space H of hypotheses, using observed data D.
- In order to do that, we need to assume a prior probability distribution over the class H.
- $P(h|D)$, the probability of the hypothesis h given data D:

$$\Pr(h|D) = \frac{\Pr(D|h)\Pr(h)}{\Pr(D)}$$

- *Maximum a posteriori hypothesis*:

$$h_{MAP} = \operatorname*{argmax}_{h \in H} \Pr(h|D) = \frac{\Pr(D|h)\Pr(h)}{\Pr(D)} = \operatorname*{argmax}_{h \in H} \Pr(D|h)\Pr(h)$$

  because the data D does not change, $\Pr(D)$ doesn't change, so the h_{MAP} is not affected by it.

- What we have been doing so far with *Maximum Likelihood Estimation* (MLE) is $h_{ML} = \operatorname{argmax}_{h \in H} \Pr(D|h)$. Assuming a uniform distribution of probabilities over the hypothesis space (i.e., assuming $\Pr(h)$ is constant) MLE and MAP are equivalent.
- Two notions of probabilistic concepts:
  (a) *Learning probabilistic concepts*: This includes learning the probability that some point belongs to a specific class.
  (b) *Bayesian learning*: Using probabilistic criterion to select a hypothesis. The hypothesis itself can be deterministic, but the criterion is inherently probabilistic.

- Issue: If we don't a lot of data, we cannot estimate $\Pr(x_1, x_2, ..., x_d|y)$ reliably. There are $k(2^d - 1)$ of these probabilities.

- *Naive Bayes Assumption*: Introduced to mitigate the problem of Bayes needing large data sets, assumes $\Pr(x_1, ..., x_d|y) = \Pr(x_1|y)\Pr(x_2|y) \cdot ... \cdot \Pr(x_d|y)$. Given this assumption, all the data we need is the priors $\Pr(y)$ for all possible labels and, for all i, $\Pr(x_i|y)$.

- Decision rule of Naive Bayes classifier: $h_{NB}(x) = \text{argmax}_y \Pr(y)\Pr(x_1, ..., x_d|y) = \text{argmax}_y \Pr(y) \prod_j \Pr(x_j|y)$.

- In general, given training data $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$:

$$\begin{aligned} h_{NB} &= \underset{h}{\text{argmax}} \prod_{i=1}^{m} \Pr((x^{(i)}, y^{(i)})|h) \\ &= \underset{h}{\text{argmax}} \prod_{i=1}^{m} \Pr(x^{(i)}|y^{(i)}, h)\Pr(y^{(i)}|h) \\ &= \underset{h}{\text{argmax}} \prod_{i=1}^{m} \left\{ \Pr(y^{(i)}|h) \prod_j \Pr(x_j^{(i)}|y^{(i)}, h) \right\} \\ &= \underset{h}{\text{argmax}} \left\{ \sum_{i=1}^{m} \log \Pr(y^{(i)}|h) + \sum_{i=1}^{m} \sum_{j=1}^{d} \log \Pr(x_j^{(i)}|y^{(i)}, h) \right\} \end{aligned}$$

- Possible issues: Features may not be conditionally independent given labels. We also might not have enough training data (can be addressed by smoothing).

- Smoothing:

$$\Pr(w|\text{Spam}) = \frac{\text{Count}(w, \text{Spam})}{\text{Count}(\text{Spam})} \quad \Pr(w|\text{Spam}) = \frac{\text{Count}(w, \text{Spam}) + 1}{\text{Count}(\text{Spam}) + |\text{Vocabulary}|}$$

- *Discriminative models*: Goal is to learn directly how to make predictions, that is, find $h : X \to Y$ by looking directly at $\Pr(Y|X)$. Discriminative models give $\Pr(\mathcal{C}_i|x)$.

- *Generative models*: Explicitly model how instances in each category are generated. For example, Naive Bayes is a generative model. Idea is to learn $\Pr(X|Y)$ and $\Pr(Y)$ and derive $\Pr(Y|X)$ using Bayes rule. Generative models give $\Pr(x, \mathcal{C}_i)$.

- In *unsupervised learning*, we only observe the input distribution $\Pr(x)$. MLE can be used in unsupervised learning by the law of total probability:

$$\underset{\theta}{\operatorname{argmax}} \Pr(X|\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{y} \Pr(X, Y|\theta)$$

- *Expectation Maximization Algorithm*: Solves above equation by iteratively updating θ. Guess the probability that a given data point has a specific label; weigh according to this probability and get best θ; compute the likelihood of the data given this model; reestimate the initial probability guesses to maximize likelihood in next iteration.

17 Hidden Markov Model

- Idea: The output label is dependant on its neighbors in addition to the input. A sequence model of transitive states. States are hidden, but each state transition comes with an observation.

- History-based model:

$$\Pr(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} \Pr(x_i | x_1, ..., x_{i-1})$$

- *First-order Markov assumption*:

$$\Pr(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} \Pr(x_i | x_{i-1})$$

  Requires knowledge only of initial state probabilities and state transition probabilities.

- Hidden Markov Model: Each state stochastically emits an observation. States are not observed themselves.

- Three questions we can ask of a HMM:

  (a) Given an observation sequence $x_1, x_2, ..., x_n$ and a model (π, A, B), where A is the transition probabilities matrix, and B is the emission probabilities matrix, how to effectively calculate the probability of a particular sequence of observations?

  (b) How to efficiently calculate the most probable state sequence given a sequence of observations? (Inference)

   (c) How to calculate (π, A, B) given many observation sequences? (Learning)

- *Viterbi algorithm*: Greedy algorithm for determining most likely sequence of states given a sequence of observations $(x_1, ..., x_n)$ we want to predict $(y_1, ..., y_n)$:

   (a) For each state s, calculate $\text{score}_1(s) = \Pr(s, x_1) = \Pr(s)\Pr(x_1|s)$.

   (b) From then on, go through all i, and for every s determine

$$\text{score}_i(s) = \max_{y_{i-1}} \Pr(s|y_{i-1})\Pr(x_i|s)\text{score}_{i-1}(y_{i-1})$$

   (c) Final state:

$$\max_y \Pr(y, x|\pi, A, B) = \max_s \text{score}_n(s)$$

- *Learning HMM parameters*: Two options

   (a) *Supervised learning with complete data*: Given a dataset of sequences of observations labeled with states, find best HMM such that (π, A, B) maximizes likelihood. Use maximum likelihood $(\pi, A, B) = \max_{\pi,A,B} \Pr(D|\pi, A, B)$, given $D = \{(x^{(i)}, y^{(i)})\}$. Usually, priors and smoothing are used. Maximum likelihood tends to work best in this case.

   (b) *Unsupervised learning with incomplete data*: We are only given a collection of observation sequences.

18 Neural Network

- At each node *input function* is used to compute the input value of all the feeding nodes.

- *Activation function* transforms this input function into a final output value (typically non-linear, e.g. sigmoid). Possible activation functions:

  - Linear unit: z
  - Threshold/sign unit: $\text{sgn}(z)$
  - Sigmoid unit: $[1 + \exp(-z)]^{-1}$
  - Rectified linear unit: $\max(0, z)$

    - Tanh unit: $\tanh(z)$

- To train, use stochastic gradient descent: Given a training set $S = \{(x_i, y_i\}$, initialize parameters w; go through epochs; For each epoch, shuffle the training set; for each training example (x_i, y_i) treat the data as the entire dataset and compute the gradient of the loss function; update based on loss gradient and learning rate. Use loss function $L = \frac{1}{2}(y - y^*)^2$.

- *Backpropagation*: Defines good way to find the gradient. The backpropagation algorithm lets us recycle precomputed partial derivatives in later calculations. We start computing at upper layers, and compute for lower layers later.

- *Recurrent neural networks*: How to deal with varying size input? Feed the output of the output of the neural net for the first set of parameters and the new parameters back into the neural net and compute. RNN are successful at modeling hidden state dependencies (like HMM), but cannot model long-distance dependencies of the hidden states.