CIS 520 - Machine Learning Notes

Eric Oh - University of Pennsylvania

September 29, 2017

1 Local Learning

The class of local learning methods seems less like learning and more like pure memorization - and in some ways it is. Main idea: given a new example x, find the most similar training example(s) and predict a similar output. We generally assume some measure of similarity or distance between examples and learn a *local* method in a neighborhood of the new example. Consider the one dimensional regression problem:

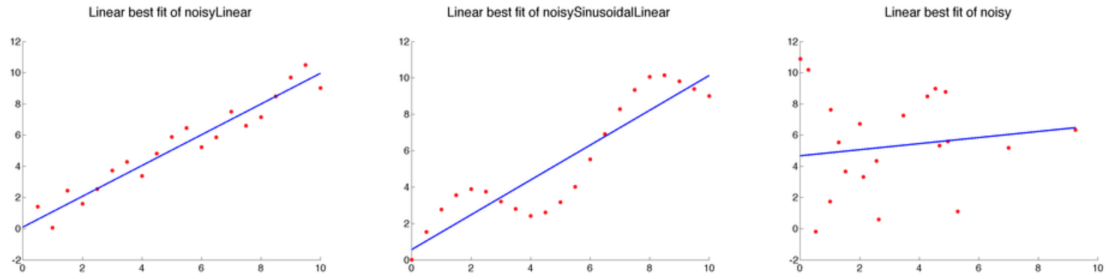


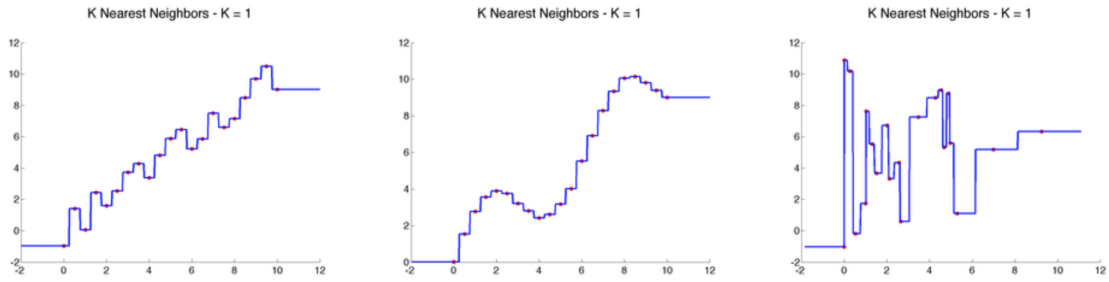
Figure 1: Clearly, linear model doesn't fit data well always

1.1 Nearest Neighbor

We could try adding higher-order polynomial terms or try a local method like nearest neighbors:

```
1 - Nearest Neighbor Algorithm
  1. Given training data D={x_i,y_i}, distance function d(.,.), and input x
  2. Find j = arg min_i d(x,x_i) and return y_j
```

Some examples of the above algorithm with $d(x, x_i) = |x - x_i|$ as x varies:



Generally, we use some arbitrary distance function to define nearest neighbor. Some common choices are the L_1, L_2, or the L_∞ norms.

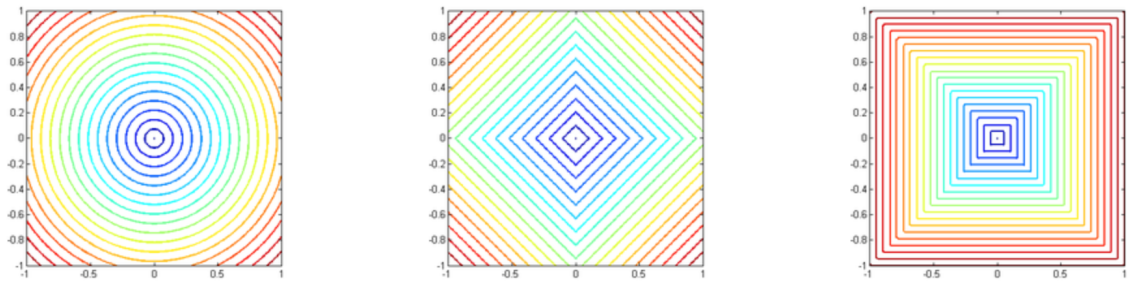
Definition 1.1. ***Norm*** *- for all $a \in \mathbb{R}$ and all $u, v \in V$,*

- $L_p(av) = |a|L_p(v)$
- $L_p(u + v) \leq L_p(u) + L_p(v)$ - triangle inequality or subadditivity
- *If $L_p(v) = 0$ then v is the zero vector*

Example 1.1. *L_p norm :* $\left(\sum_j |x_j|^p\right)^{\frac{1}{p}}$

Definition 1.2. L_0 ***pseudo-norm*** : $|x|_0$ = *number of* $x_j \neq 0$

The contours of the most common distance measures are below.



L_2*norm* L_1*norm* L_{inf}*norm*

Note that for the L_p norm, p must be at least 1. L_p norms with $p < 1$ result in **concave** contours - NOT GOOD. So how do norms delate explicitly to distances?

$$d_p(x, y) = |x - y|_p$$

Note that norms are **SCALE-INVARIANT**. Thus, if x is a vector of non-negative terms, then

$$\sum_i x_i \qquad \text{and} \qquad \sum_i i x_i$$

are both norms.

1-Nearest Neighbor is one of the simplest examples of a **non-parametric** method (ie. model structure determined by the training data). Non-parametric models are much more flexible and expressive than parametric models, and thus **overfitting** can be a big concern. One nice property of NP models is that since the complexity increases as the data increases, with enough data, NP models can model nearly anything and achieve close to zero bias for any distribution (ie. consistent..sort of). A huge problem with this, however, is that for any finite sample size there will likely be huge variance (ie. overfitting).

One way to reduce the variance is local averaging : instead of just one neighbor, find K and average their predictions.

```
K-Nearest Neighbors Algorithm
  1. Given training data D={x_i,y_i}, distance function d(.,.), and input x
  2. Find {j_1,...,j_K} closest examples wrt d(x,.)
      a. (Regression) if y \in \mathbb{R}, return average of y_{j_k}
      b. (Classification) if y is 1 or -1, return majority
```

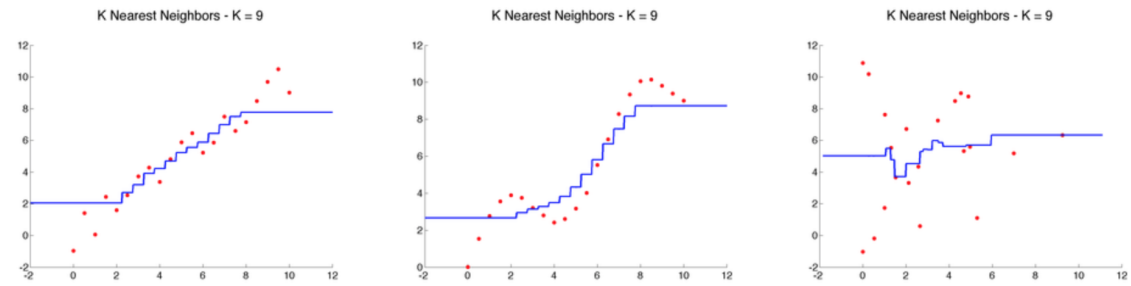


Figure 2: 9 nearest neighbors - smoother prediction but edges still sketchy

1.2 Kernel Regression

Two main shortcomings of K-NN method:

1. All neighbors receive equal weight
2. Number of neighbors chosen globally

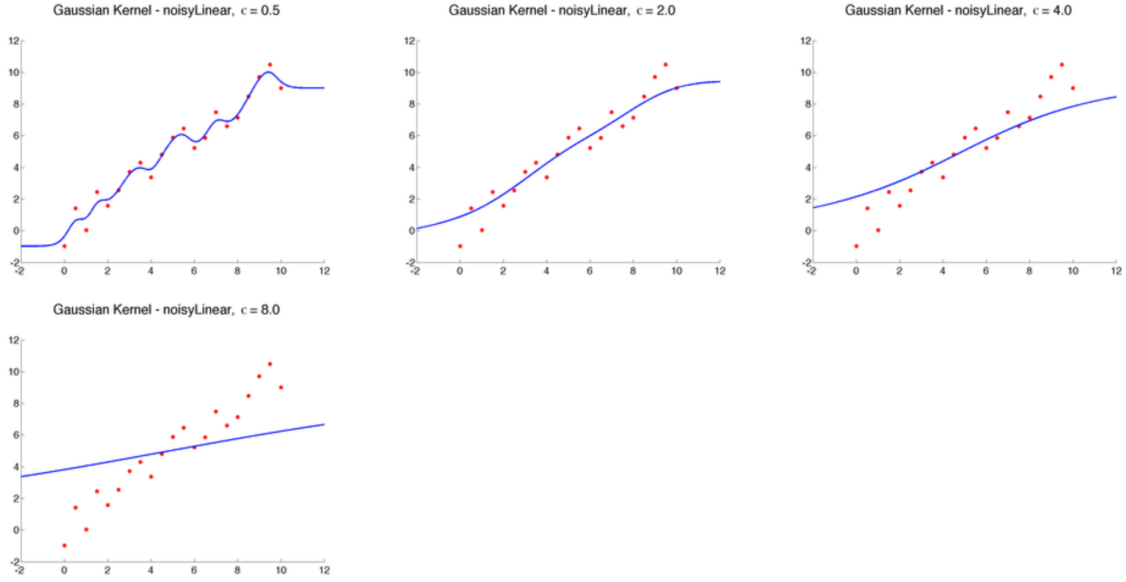
Kernel regression address these issues by *using all neighbors* but with different weights (ie. closer neighbors receiving higher weights and vice versa). Weighting function is a **kernel** and it measures similarity (as opposed to distance) between examples. Can convert from a distance d(.,.) to kernel K(.,.) easily - most common way is via Gaussian kernel (ignoring normalizing constants):

$$K(x, x_i) = \exp\left(\frac{-d^2(x, x_i)}{h}\right)$$

Kernel Regression/Classification Algorithm

1. Given training data $D = x_i, y_i$, Kernel function K(.,.), and input x
   - (Regression) if $y \in \mathbb{R}$, return weighted average: $\sum_i y_i \frac{K(x,x_i)}{\sum_j K(x,x_j)}$
   - (Classification) if $y \in \pm 1$, return weighted majority: $\text{sgn}\left(\sum_i y_i K(x, x_i)\right)$

One of the keys kernel regression is h, the **bandwidth**, which determines how quickly the influence of neighbors falls off with distance. The following shows the effect of increasing levels of h.



As $h \to \infty$, all the neighbors weight the same and the prediction is the global average.
As $h \to 0$, prediction tends to 1-NN.
Clearly choosing the "right" h is very important - in practice, usually use **cross-validation** to pick.

2 Decision Trees

- Gives nice interpretable output but each split uses exponentially less data (observations)
- How do we decide which features to split on? How to define importance? Need definition of information

2.1 Entropy

Suppose X can have one of m values, $V_1, \ldots, V_m$, with $P(X = V_i) = p_i$. Entropy is the smallest possible number of bits, on average, per symbol, needed to transmit a stream of symbols drawn from X's distribution.

Definition 2.1. *Entropy :* $H(X) = -\sum_{j=1}^{m} p_j \log_2 p_j$

- High entropy : X is from uniform (boring) distribution. Values sampled from it would be all over the place.
- Low entropy : X is from varied (valleys and peaks) distribution. Values sampled from it would be more predictable (from the peaks).

Entropy is the expected value of the information content (surprise) of the message $\log_2 p_j$. Range of entropy is from 0 to ∞.

Example 2.1. *1. If an event is certain, entropy is 0*

2. *If two events are equally likely, entropy is 1* $(-(-0.5 * \log(0.5) + -0.5 * \log(0.5))$

Definition 2.2. *Specific Conditional Entropy :* $H(Y|X = v)$*, the entropy of* Y *among only those records in which* X *has value* v*.*

Suppose I'm trying to predict output Y and I have input X

X = College Major

Y = Likes "Gladiator"

X	Y
Math	Yes
History	No
CS	Yes
Math	No
Math	No
CS	Yes
History	No
Math	Yes

Let's assume this reflects the true probabilities

E.G. From this data we estimate

- *P(LikeG = Yes) = 0.5*
- *P(Major = Math & LikeG = No) = 0.25*
- *P(Major = Math) = 0.5*
- *P(LikeG = Yes | Major = History) = 0*

Note:

- *H(X) = 1.5*
- *H(Y) = 1*

Definition 2.3. *Conditional Entropy :* $H(Y|X) = \sum_j P(X = v_j)H(Y|X = v_j)$*, the average specific conditional entropy of* Y

- *H(Y|X=Math) = 1*
- *H(Y|X=History) = 0*
- *H(Y|X=CS) = 0*

Definition 2.4. *Information Gain :* $IG(Y|X) = H(Y) - H(Y|X)$*, how many bits on average would it save me in transmitting* Y *if both ends of the line knew* X*?*

Example 2.2. *Suppose,*
$H(Y) = 1, H(Y|X) = 0.5$*. Then* $IG(Y|X) = 1 - 0.5 = 0.5$*, and knowing* X *gives gain of 0.5 information (unit is bits).*

Note: range of IG is from 0 to ∞. If Y and X are independet, then $IG = 0$.
In general, features with more options (more categories) have more possible information gain than binary features.

Key to Decision Trees: How do we prevent overfitting? How we know when to stop splitting?! Number of leaves grows exponentially

3 MLE

Self-explanatory.

3.1 Learning Guarantees

The following bound will be useful (based on Hoeffding inequality):

Theorem 3.1. $P(|\hat{\theta}_{MLE} - \theta| \geq \epsilon) \leq 2e^{-2n\epsilon^2}$

In other words, MLE deviates from true parameter exponentially with n.

4 MAP (maximum a posteriori)

Bayes rule gives

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

Definition 4.1. $\hat{\theta}_{MAP} = arg\ max_\theta P(\theta|D) = arg\ max_\theta \log(P(D|\theta)) + \log(P(\theta))$

Note: the $\log(P(\theta))$ acts as a regularizer for the log likelihood for MAP estimation.
Note: as $n \to \infty$, the prior's effect vanishes and we recover the MLE.
Note: MAP is the simplest Bayesian approach to parameter estimation - sets estimate equal to mode of posterior distribution $P(\theta|D)$. There is more information in the posterior such as the mean and variance of the distribution.

5 Regression

5.1 Ridge Regression

If we have too many features compared to data points, we want to do some regularization. Ridge regression does this with an L_2 penalization. For instance

$$W = (X^T X)^{-1} X^T Y$$

wouldn't work with too many features (ie. $(X^T X)$ not invertible). Thus we minimize

$$\|y - X\beta\|_2 + \lambda\|\beta\|_2$$

which eseentially finds $W = (X^T X + \lambda I)^{-1} X^T Y$. The addition of the penalty makes the matrix positive semi definite and invertible.

6 Classification

Goal: Learn to predict a categorical outcome from input and minimize expected classification error:

$$E(x, y)[I(h(x) \neq y)]$$

If we had binary data (0's and 1's), we could fit linear regression and just predict everything > 0.5 to be 1 and everything < 0.5 to 0. We **CAN** fit linear regression to binary data, but this is bad!! This gives probabilities below 0 and above 1!!

Make a transformation! Most common one is **logit** function:

$$f(s) = \frac{1}{1 + e^{-s}}$$

6.1 Generative v. Discriminative Approaches

One of the main distinctions between classification algorithms is the distinction between Generative and Discriminative. Essentially,

- Generative classifiers model $P(x, y)$
- Discriminative classifiers model $P(y|x)$

Generative Approach Estimate $P(\hat{x}, y)$ from training data (ie. MLE or MAP) and then produce classifier:

$$h(x) = \text{argmax}_y P(\hat{y}|x) = \text{argmax}_y \frac{P(\hat{x}, y)}{\sum_{y'} P(\hat{x}, y)}$$

For binary classification, this reduces to $h(x) = \left\{ \begin{array}{ll} 1, & \text{if } P(\hat{x}, 1) > P(\hat{x}, -1) \\ 0, & \text{otherwise} \end{array} \right\}$

If we are **very** lucky and learn a perfect model for $P(x, y)$, then we are **Bayes optimal**. But usually never happens. One example of a generative model is **Naive Bayes**.

Discriminative Approach Joint distribution $P(x, y)$ is hard to model, so focuses on $P(y|x)$.

- Logistic regression models $P(y|x)$ as logit-linear and then estimates params using MLE or MAP
- SVM and boosting learn a function $h(x)$ that minimizes expected classification error

6.2 Logistic Regression

Thus we obtain something like (for Y=1 and Y=-1)

$$P(Y=1|x,w) = \frac{1}{1+\exp(-w^T x)} = \frac{1}{1+\exp(-yw^T x)}$$
$$P(Y=-1|x,w) = 1 - P(Y=1|x,w) = \frac{\exp(-w^T x)}{1+\exp(-w^T x)} = \frac{1}{1+\exp(-yw^T x)}$$

Log-likelihood for logistic regression can be written

$$\log(P(D_y|D_x,w)) = \log\left(\prod_i \frac{1}{1+\exp(-y_i w^T x_i)}\right) = -\sum_i \log(1+\exp(-y_i w^T x_i))$$

Since log-likelihood for logistic regression is concave downwards (or convex upwards), can solve using **gradient ascent**!

Start at some initial point $w^0 = 0$ and climb up in the steepest direction defined by the gradient of our objective function. The gradient ascent update is

$$w^{t+1} = w^t + \nu_t \nabla_w l(w)$$

where ν_t is the update rate and $\nabla_w l(w) = \sum_i y_i x_i (1 - P(y_i|x_i, w_i))$

Why is the decision bounary linear?
Consider the condition that holds at the boundary:

$$P(Y=1|x,w) = P(Y=-1|x,w) \Rightarrow \frac{1}{1+\exp(-w^T x)} = \frac{\exp(-w^T x)}{1+\exp(-w^T x)} \Rightarrow w^T x = 0$$

Solving that system gives a line!!

Computing MAP
We can assume a prior distribution on the parameters w that we are trying to estimate. Assume

$$w_j \sim \mathrm{N}(0,\lambda^2) \rightarrow P(w) = \prod_j \frac{1}{\lambda\sqrt{2\pi}} \exp\left(\frac{-w_j^2}{2\lambda^2}\right)$$

giving an objective function of

$$\mathrm{argmax}_w \log(P(w|D,\lambda)) = \mathrm{argmax}_w (l(w) + P(w|\lambda)) = \mathrm{argmax}_w \left(l(w) + \frac{1}{2\lambda^2} w^T w\right)$$

We can solve for w similarly using gradient ascent. Note that the effect of the prior is to **penalize** large values of w.

Can also do the same as above with multinomial outcomes!

$$P(Y=k|x,w) = \frac{\exp(w_k^T x)}{1+\sum_{k'=1}^{K-1}\exp(w_{k'}^T x)} \qquad k = 1,\ldots,K$$

6.3 Naive Bayes

Naive Bayes classifier is an example of a **generative** model : we model $P(x,y) = P(y)P(x|y)$ where

- $P(y)$: class prior
- $P(x|y)$: class model

We estimate them separately as $\hat{P(y)}$ and $\hat{P(x|y)}$. Note: the class prior is different than the parameter prior classic Bayesian analysis. We then use our estimates to output a classifier using Bayes rules:

$$h(x) = \mathrm{argmax}_y \hat{P(y|x)} = \mathrm{argmax}_y \frac{\hat{P(y)}\hat{P(x|y)}}{\sum_y \hat{P(y)}\hat{P(x|y)}} \propto \mathrm{argmax}_y \hat{P(y)}\hat{P(x|y)}$$

In English: Classify a new instance x based on a tuple of attribute values $x = (x_1, \ldots, x_n)$ into one of the classes $y_j \in Y$.

Estimating $P(y)$: Can be estimated from the frequency of classes in the training examples

Estimating $P(x|y)$: The *Naive Bayes assumption* is that $P(x|y) = \prod_i P(\hat{x_i}|y)$ (Conditional independence)

If we assume both Y and X are discrete, can do (MLE estimation)

$$P(\hat{y}) = \frac{N(y = y_j)}{N} \qquad P(\hat{x_i}|y) = \frac{N(x_i = x_j, y = y_j)}{N(y = y_j)}$$

Note: If Y and/or X are continuous, we can pick some model for the distribution (ie. Gaussian).

Problem with MLE though is that if we see no training examples for some class, we have 0 probabilities! To deal with this, we can smooth the probabilities to avoid overfitting.

$$P(\hat{x_i}|y) = \frac{N(x_i = x_j, y = y_j) + mP_{i,k}}{N(y = y_j) + m}$$

This is sort of like **empirical bayes** where our prior includes information on probabilities in the entire sample?

Note for implementation: DO NOT MULTIPLY PROBABILITIES. THIS LEADS TO UNDERFLOW. INSTEAD ADD LOG PROBABILITIES.

6.4 MAP

To take care of the 0 probability problem with MLE, just add a prior to each x_j (Ie. for counts of words in topic documents, adding Beta$(0.5, 0.5)$ is a Jeffrey's prior, adding Beta$(1, 1)$ is Laplace smoothing). BUT THIS KIND OF SUCKS. We know "the" is more common, so instead we could use an **empirical bayes** thing where the prior is proportional to how often we see in data.

6.5 Naive Bayes vs Logistic Regression

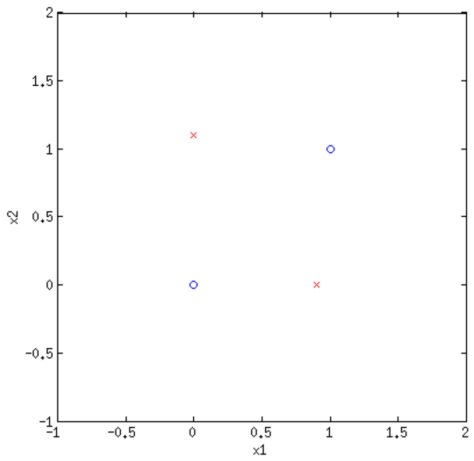
Model	Naive Bayes	Logistic Regression
Assumption	$P(\mathbf{X}\|Y)$ is simple	$P(Y\|\mathbf{X})$ is simple
Likelihood	Joint	Conditional
Objective	$\sum_i \log P(y_i, \mathbf{x}_i)$	$\sum_i \log P(y_i \mid \mathbf{x}_i)$
Estimation	Closed Form	Iterative (gradient, etc)
Decision Boundary	Quadratic/Linear (see below)	Linear
When to use	Very little data vs parameters	Enough data vs parameters

- If we only care about probabilities, probably better to use logisitic regression.
- If we only care about which class is most likely, might be better to use Naive Bayes.

If variance of the classes are wildly different, there can be a non-linear boundary for Naive Bayes. If we assume that $P(X_j|Y)$ is Gaussian and Y is binary and $P(X_j|Y)$ have same variance for both classes, then can show that it is equivalent to logistic regression.

7 Basis Functions

We can still learn difficult, non-separable datasets by transforming the inputs using **basis functions**. Can create new, good features. Consider the XOR dataset



One basis that works quite well is **radial basis functions (RBF)**.

Definition 7.1. ***RBF*** *is a real-valued function whose values depends only on the distance between the point and some "center" c :*

$$\phi(x, c) = \phi(\|x - c\|)$$

Common functions include the Gaussian and various polynomials. Often need to cross-validate to pick the best hyperparameters.

Example 7.1. *For Gaussian basis,* $\exp\left(\frac{(x-\mu_K)^2}{c}\right)$*, need to pick best number of centers* K *and the width* c *for each.*

To find the locations of the centers, standard RBF will use K means clustering to find the K clusters and put the K centers there.

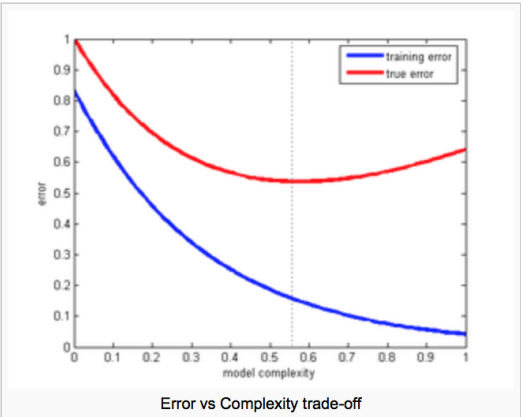
Note: RBF's are **NOT SCALE INVARIANT**. Gaussian basis assumes that all features are of equal importance. Any time we use some distance (ie. in a Kernel), there is this implicit assumption.

8 Overfitting and Regularization

Overfitting happens when a model learns to describe noise in addition to the real dependencies between input and output. In terms of bias/variance decomposition, a very complex model can have small error on any dataset (training examples) but large error on new examples (test data) due to the variance being too high.

Example 8.1.

- *For RBFs, number of centers and width of gaussians measures complexity (increase number of centers and make width smaller to increase complexity)*
- λ *in ridge regression measures complexity* $\left(\frac{1}{\lambda}\right)$
- K *in KNN measures complexity* $\left(\frac{1}{K}\right)$
- *Depth of tree in Decision Trees is one way to measure complexity*



Error vs Complexity trade-off

8.1 Adjusting complexity penalty using Cross-Validation

ASSUMES I.I.D DATA. HAVE TO BE VERY CAREFUL FOR TIME SERIES.

Definition 8.1. ***Leave one out CV (LOOCV)*** *: For each complexity setting, learn a classifer on* all the training data except one example *and evaluate its error on the remaining example. Requires building n models....very slow! Only really used for datasets with less than 100 examples.*

Leave one out Error:

$$\frac{1}{n}\sum_i I(h(x_i; D_{-1}) \neq y_i)$$

where D_{-1} is the dataset minus the i^{th} example and $h(x_i; D_{-1})$ is the classifier trained on that dataset.

Definition 8.2. ***K-fold CV*** *: Split the data into K subsets or folds (usually $K = 10$) $D_1, \ldots, D_k$ and leave out each fold in turn for validation.*

K-fold Validation Error:

$$\frac{1}{n}\sum_{k=1}^{K}\sum_{i\in D_k} I(h(x_i; D - D_k) \neq y_i)$$

where $D - D_k$ is the dataset minus the k^{th} fold and $h(x_i; D - D_k)$ is the classifier trained on that dataset.

Note, often in industry, separating data into the development and validation sets is time dependent. We want to predict home prices/ stock prices/ etc. in the **FUTURE**. So we might train on data from 1996-2016 and then get validation error on 2017 data. (can't just take a simple random sample).

If we have HUGE data (ie. Google, Facebook), can probably just split data 50-50 to train and test.

8.2 Bias/Variance Decomposition

Can show that error $= E[(y - \hat{y})^2] = \text{Bias}(\hat{y})^2 + \text{Var}(\hat{y}) + \sigma^2(noise)$.

9 Regression Penalties and Priors

Review:

- Given a set of observations with labels y
- Generate features, x, for each observation (hardest, longest part)
- Learn a regression model to predict y, $y = \beta x$; most of the β are 0 though!!

Two Interpretations of regression:

1. Minimize (penalized) squared error
2. Maximize likelihood

Note: OLS (MLE) minimzes bias (doesn't care about overfitting). Ridge regression (or MAP) minimzes bias AND variance (penalizes non-important features).

Different Norms, different penalties:
Minimize penalized error: $\|y - \beta x\|_2^2 + \lambda f(\beta)$

- $\|\beta\|_2$ - L_2 makes all the β_j a little smaller (RIDGE)
- $\|\beta\|_1$ - L_1 drives some β_j to 0 (LASSO or LARS)
- $\|\beta|_0$ - L_0 counts number of nonzero β_j - "stepwise regression"

all of the above encourage β_j to be smaller, ie. shrink β. Bigger λ gives more shrinkage!! L_2 and L_1 are nice because they are convex!! Can also do the **elastic net**:

$$\text{argmin}_\beta \|y - \beta x\|_2^2 + \lambda_1\|\beta\|_1 + \lambda_2\|\beta\|_2$$

9.1 Feature selection for regression

Goal: minimize error for a test set
Approximation: minimize a penalized training set error
Note: always better to get approximation to correct loss function than get exact answer to wrong loss function!!

Different regularization priors

- L_2 corresponds to Gaussian prior: $p(\beta) = \exp\left(-\frac{\|\beta\|_2}{\sigma^2}\right)$ - not scale invariant?
- L_1 corresponds to Laplace prior: $p(\beta) = \exp\left(-\frac{\|\beta\|_1}{\sigma^2}\right)$ - not scale invariant like OLS
- L_0 corresponds to Spike and Slab, assumes some are 0?

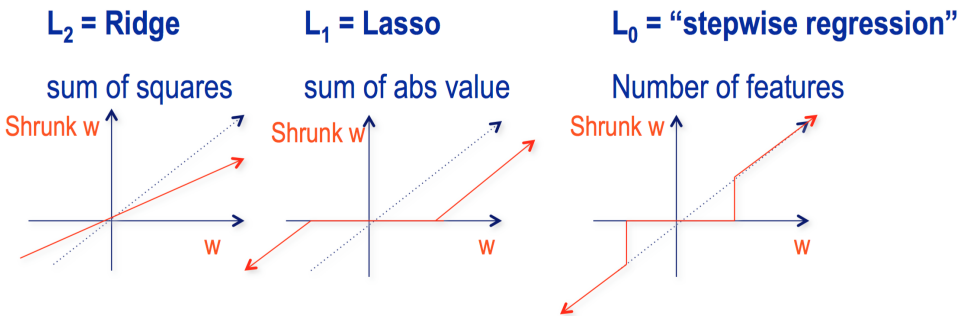
Solving regularization penalties:

- L_2 - $\beta = (X^T X + \lambda I)^{-1} X^T y$
- L_1 - convex optimization, gradient descent
- L_0 - search (stepwise or streamwise)

Properties:

- L_0 norm most strongly encourages weights to be set to 0.
- L_1 and L_0 can handle exponentially more features than observations ; L_2 cannot
- L_2:all β shrink by constant factor
- L_1:all β shrink by constant amount (if β lower than amount shrunk, then goes to 0. Determined by λ). Can be not great because to make a lot of features 0, must shrink all features by A LOT. (Does shrinkage and zeroing in "one" step).
- L_0 throws out all the non important features, leaves others?

If x's have been standardized, can visualize the shrinkage:



9.2 Streamwise Regression

1. Initialize :
   - model $= \{\}$, $Err_0 = \sum_i (y_i - 0)^2 + 0$
2. For each feature x_j:
   - Try adding feature x_j to the model
   - **if** $Err = \sum_i (y_i - \sum_j \beta_j x_{ij})^2 + \lambda \|\beta\|_0 < Err_{j-1}$, then accept new model and set $Err_j = Err$.
   - **else** Keep old model and set $Err_j = Err_{j-1}$

The above algorithm is **extremely** greedy!! It is also biased to which features we try first...not great.

9.3 Stepwise Regression

1. Initialize :
   - model = {}, $Err_0 = \sum_i (y_i - 0)^2 + 0$
2. Repeat (up to p times):
   - Try adding feature x_j to the model
   - Pick the feature that gives the lowest error
   - Calculate new error with that feature in it
   - **if** $Err < Err_{old}$, add the feature to the model, $Err_{old} = Err$
   - **else** Halt

9.4 Stagewise Regression

Like stepwise, but at each iteration, keep all of the coefficients β_j from the old model but just regress the residual on the new candidate feature β_j.

10 Minimum Description Length

For L_1 and L_2 penalties, can pick λ using cross-validation. How do we pick complexity for L_0 norm, which penalizes number of parameters??

MDL

- Sender and receiver both know X
- Wnat to send y using minimum number of bits
- Send y in two parts
  1. Code (the model)
  2. Residual (training error = in sample error)

Example 10.1. ***Decision Tree***

- *Code = the tree*
- *Residual = the misclassifications*

Example 10.2. ***Linear regression***

- *Code = the weights*
- *Residual = prediction errors*

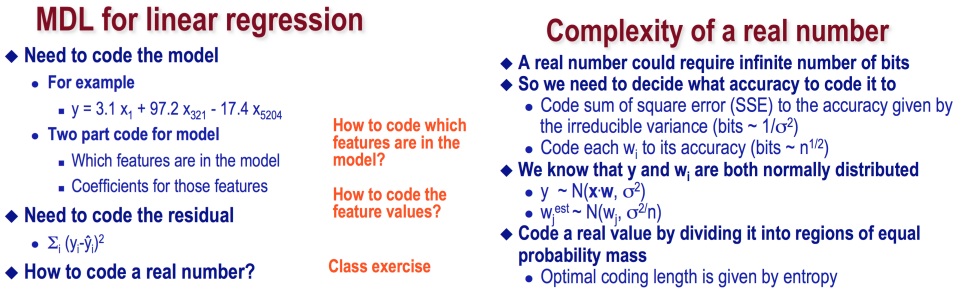
Example 10.3. Complexity of classification *If* $y = (0, 1, 1, 0, 1, 0)$*, then since* $P(Y = 1) = P(Y = 0)$ *entropy is high.*
If 1's are very rare, ie. $y = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, \ldots)$*, then entropy*

$$-\sum \frac{1}{100} \log\left(\frac{1}{100}\right) + \frac{99}{100} \log\left(\frac{99}{100}\right)$$

Since $\frac{99}{100}$ *is nearly 1, need roughly* $\frac{1}{100}$ *bits. And as n gets larger and larger and if 1 is still rare, only* $\log(P(Y = 0)$ *determines number of bits.*

Example 10.4. *y = (0,1,1,0,1,1,1,0,0)*
$\hat{y}$*=(0,0,1,1,1,1,1,0,0)*
$y - \hat{y}$*=(0,1,0,-1,0,0,0,0,0)*

We can code 1 and -1 the same because all we need to know if correct or wrong classification.



10.1 MDL linear regression

Code the residual/data log-likelihood

$$\log(\text{likelihood}) = n\log(\sqrt{2\pi}\sigma) + \frac{1}{2\sigma^2}\exp(-\|y-\hat{y}\|)_2^2$$

Code the model

- For each feature, is it in the model?
- If it is included, what is its coefficient?

Choosing models to code the residuals and model **is exactly** the bias, variance tradeoff!! If we fit a super complex data, need less bits to code residual (smaller bias) but variance will be higher so need more to code the model. If we fit a super simple mode, need more bits to code residual (larger bias) but variance will be smaller so need less to code the model.

But we don't know σ^2, the irreducible error

- Option 1 - use estimate from previous iteration ;

$$\sigma^2 = \text{Err}_{q-1}$$

- Option 2- use estimate from current iteration ;

$$\sigma^2 = \text{Err}_q$$

10.2 Complexity of features

If you expect q features in the model, each will come in with probability $\frac{q}{p}$ (p total features under consideration). The total cost **for all p features** is then

$$p(-(q/p)\log(q/p) - ((p-q)/q)\log((p-q)/p))$$

Note: we multiply by p at the beginning because we the total cost is for all p features, so pay for all of them. $-(q/p)\log(q/p)$ is for features in the model and $(p-q)/q)\log((p-q)/p)$ is for features not in the model.

Note: complexity of model increases logarithmically with number of features. AND THIS COSTS US IN BITS! To make it worth it we must also be lowering number of bits needed to code residaul (ie. lower bias)

Example 10.5.

- *If* $\frac{q}{p} = \frac{1}{2}$, *total cost is* p *; cost/feature* $= 2bits$
- *If* $\frac{q}{p} = 1$, *total cost is* $\log(p)$ *; cost/feature* $= 1bit$

Regression Penalty Methods Minimize $\frac{\text{Err}_q}{2\sigma^2} + \lambda\|\beta\|_0$ (L_0 penalty on coefficients, SSE with the $\|\beta\|_0 = q$ features)

Table 1: My caption

Method	Penalty	
AIC	1	code coefficient using 1 bit
BIC	$(1/2)\log(n)$	code coefficient using $n^{1/2}$ bits
RIC	$\log(p)$	code feature presence/absence; prior:one feature will come in

AIC is too cheap!! It will probably overfit and add too many features (but if you have a ton of features, might be a good bet). When $n > p$ and not too many features, BIC is probably a good bet. Often in ML, we are in RIC land.

Example 10.6.

- *You expect 10 out of 100,000 features, n=100 - RIC*
- *You expect 200 out of 1,000 features, n=1,000,000 - BIC*
- *You expect 500 out of 1,000 features, n=1,000 - AIC*

Why does MDL work?

- We want to tune model complexity - how many bits should we use to code the model?
- Minimize: Test error = training error + penalty = bias + variance

Example 10.7.

- *You think 10 out of 100,000 features will be significant - probably use L_0 norm with RIC*
- *You think 500 out of 1,000 features will be significant - use anything **but** L_0 with RIC*

Note: RIC is quite similar to Bonferroni correction.

11 Neural Networks

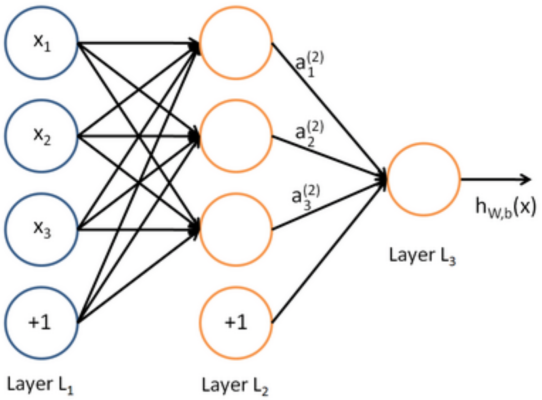
Uses **very** flexible model forms : $\hat{y} = f(x;\theta)$. Uses different loss functions such as L_2 norm, log-likelihood, logistic/softmax, etc. If we knew the functional form of the data, we wouldn't need such flexible forms used in neural nets!

- Non-parametric (or technically, semi-parametric) - flexible model form
- Used when there are vast amounts of data

Deep Learning forms:

- Supervised - convolutional, recurrent (LSTM) ; generalize logistic reg to semi-parametric form
- Unsupervised - autoencoder ; generalize PCA to semi-parametric form
- Semi-supervised
- Reinforcement Learning

Below is a simple three layer artificial neural net (ANN). It basically stacks logistic regressions.

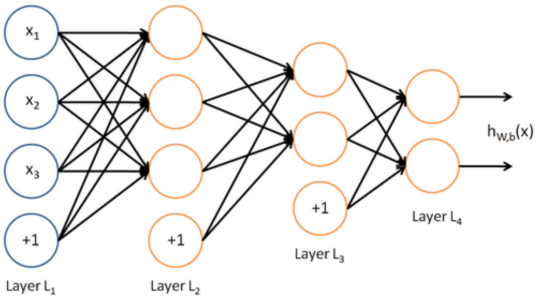


Let σ be the sigmoid (logistic) function. Then prediction could be something like

$$\hat{y} = \sigma\left[\sum_{k=1}^{3} a_k \sigma\left(\sum_{j,k} w_{j,k} x_j\right)\right]$$

Since the above is non-linear, to minimize the loss (for instance $\|y-\hat{y}\|_2^2$), we use (stochastic) gradient descent to estimate parameters and use the chain rule (backpropagation) to find derivatives. The **number of parameters** in the model above is given by *each line*, ie number of parameters is number of lines above.

Can add more hidden layers (yellow circles) to get a deep network!!



The intuition for adding more layers is that each layer is essentially doing feature detection. Each layer automatically determines features that when fed into another layer can predict whatever very well!

Definition 11.1. ***Fully connected network*** *- each input in layer is connected to the next layer*

ANNs input **percepts** to output categories or actions

- Image of object → what it is
- Board position → probability of winning
- Sequence of words in English → their Chinese translation

Classic, revolutionary paper : ImageNet Classification with Deep Convolutional Neural Networks

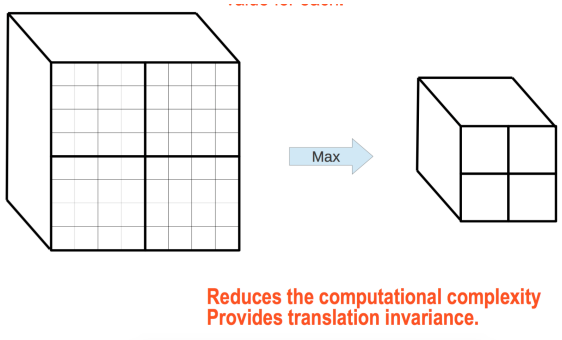
Evolution of functions:

- Logistic function - traditional, works okay
- Hyperbolic tangent - very bad, slow to train
- **Rectified Linear Unit (ReLU)** - very good, quick to train ; $f(x) = x^+ = \max(0, x)$
- Can use any nonlinear function

11.1 AlexNet

- 7 hidden layers ; 5 convolutional layers and 2 fully connected layers
- In vision, neuron may only get inputs from limited set of "nearby" neurons
- **Local/Max pooling**: partitions the input image into non-overlapping rectangles and outputs the maximum value for each. Follows 1st, 2nd, and 5th convolutional layers in AlexNet. See example below

Taking the max is intuitive because in images, we are often interested in knowing *if* something is there or not. ie. is a cat in the image?



How did AlexNet do?:

- 60 million parameters, so overfit A LOT. To combat this, generated pseudodata by shifting images slightly so that they get more examples but dont have to label more data. → DATA AUGMENTATION
- For example, thought tape player was a cellular telephone because in 2012 cell phones were much more frequent

11.2 Modern Deep Nets

- Often use ReLUs - less problems of saturation than logistic
- Use a variety of loss functions - often log likelihood (uses softmax: $P(y = j|x) = \frac{\exp(w_j^T x)}{\sum_k \exp(w_k^T x)}$)
- Can be very deep
- Solved with mini-batch gradient descent
- Regularized using L_2 penalty plus "dropout"....or partial convergence (form of regularization by stopping early somehow the weights are smaller?)

Note that softmax is often used in the final layer of NNs. Generalization of the logistic function that "squashes" a K-dim vector of arbitrary real values to a K-dim vector $\sigma(z)$ of values in the range [0,1] that add up to 1.

Definition 11.2. ***Dropout***

1. *randomly (temporarily) remove a fraction p (usually 0.5) of the nodes (w/ replacement)*
2. *Do gradient descent on the other fraction*
3. *Then repeat*

Repeatedly doing this samples (in theory) over exponentially many networks (bounces network out of local minima). For the final network, use all the weights but shrink them by p.

11.3 Stochastic Gradient Descent

With standard gradient descent, we would calculate error over **all** data points. **Stochastic gradient descent** does this for each point at a time. Ie. if $Err = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, normal GD would calculate $\frac{dErr}{d\vec{w}}$, which is over all (for example million) points. Stochastic GD calculates derivative of $Err_=(y_i - \hat{y}_i)^2$ instead which is much cheaper.

Properties:

- Stochastic GD is probably faster and bouncier - can essentially do "online" learning by updating weights as each data point comes in
- Normal GD has been proven to decrease error each iteration ; not so for stochastic GD bc we are only doing one weight at a time
- Learning rates are quite different
- FOR ALL GD stuff we know when convergence is reached by cross-validation.

Most software that does GD approximates the derivative of the error using symmetric version (doesn't actually use the chain rule):

$$\frac{dErr(w)}{dw} = \frac{Err(w+h) - Err(w-h)}{2h}$$

Note: the above is done for each weight to see how it changes the error.

In practice, nobody uses either!!! Normal GD is too slow and stochastic GD is too bouncy. They use **minibatch** GD.

Definition 11.3. ***Mini-Batch GD****: does GD batch by batch (ie. 20 points at a time). Much faster and more stable than either of the above. Mini-Batch GD has stochastic GD and normal GD as special cases (batch=1 or batch=n)*

Definition 11.4. ***Momentum****:*

$$\Delta w^t = \eta \frac{dErr}{dw} + m\Delta w^{t-1}$$

Want the hessian, but this is a compromise

11.4 Learning Rate

- Want to adjust the learning rate over time, ie. make the learning rate smaller as we get closer to local max

$$\Delta w^t = \eta(t)\frac{dErr}{dw}$$

- Could also have different learning rate for each weight - **Adagrad**: makes the learning rate depend on previous changes in each weight. Increases the learning reate for more sparse parameters and decreases the learning rate for more sparse ones.

$$\Delta w_j^t = \frac{\eta}{\|\delta w_j^\tau\|_2}\frac{dErr}{dw_j}$$