- Simple matching coefficient $\frac{(C_{11}+C_{00})}{(C_{00}+C_{01}+C_{10}+C_{11})}$
- How many times the variables are the same
- Jaccard coefficient $\frac{C_{11}}{(C_{01}+C_{10}+C_{11})}$
- intersection divided by union of 1 values

Fitting stumps using classification error

```
for every feature j
    for every object i
        threshold = X(i,j)
        find mode of y when X(:,j) > threshold
        find mode of y when X(:,j) < threshold
        classify all examples based on threshold
        sum(yhat ~= y)
        store this rule ifits the lowest error so far
```

Classifying a new example

```
function [y] = predict(model,X)
[t,d] = size(X);
if isempty(model.splitVariable)
  y = model.splitSat*ones(t,1);
else
  y = zeros(t,1);
  for i = 1:t
    if X(i,model.splitVariable) > model.splitValue
      y(i,1) = model.splitSat;
    else
      y(i,1) = model.splitNot;
```

Big O notation

- Computing one rule costs $O(n)$
- Compute scores for dt rules (where t = thresholds) is $O(ndt)$
- If we use at most n thresholds for each feature its now $O(n^2d)$
- If we sort the features it costs $O(nlogn)$ per feature and you do at most $O(n)$ updates per feature
- This gives us runtime $O(ndlogn)$ down from $O(n^d)$
- For trees of depth m,
- the number of stumps to fit is 2^{m-1} naive approach would be to say $O(2^{m-1}ndlogn)$
- However at each depth we split the n examples across the stumps. this means that each depth will only need to look at n examples and the cost for each layer is $O(ndlogn)$.
- This gives us total cost $O(mndlogn)$ to go through all m layers.

Effect of depth

- Depth will usually lead to more complicated and longer models that will also decrease the training error - this usually means you are overfitting to your training set.
- It usually best to choose the depth based on cross validation to see where the model has begun to overfit the data.

Learning theory

Training vs Test error

- Memorization vs Learning
- Can do well on training data by memorizing it, only learned if you do well in new situations
- Most common assumption: **independent and identically distributed**
  - all objects come from the same distribution
  - the objects are sampled independently (order doesn't matter)

Validation Sets and Cross Validation - train on your training data, predict on your validation data

- Validation error initially decreases, eventually increases (overfits), its only unbiased if you use it once
- if you minimize it to choose between models it introduces optimization bias
- Optimization bias is:
  - small if you compare a few models - ie. best decision tree on training set depths 1to 10
  - validation likely still approximates error, overfitting risk is low
  - large if you compare a lot of models - ie. all decision trees of depth 10 or less
  - comparing a ton of models some that may likely have low validation error by chance, overfitting is likely
- Cross Validation - ie 10 fold cross-validation
- Train on 90% of the data, validate on the other 10%, repeat this 10 times where each fold (10% of the data) gets to be the validation set once, average the score, this is a likely test error approximation
- Usually want to retrain based on the results learned from the CV (ie. after picking the best depth), and if data is ordered then folds should be random splits

Fundamental Tradeoff

- How small can you make the training error vs. (complex models - small training error, sensitive to the training data)
- How well training error approximates the test error (simple models - good approximation of test error but doesn't fit the training data well)
- Irreducible Error that you cannot lower
- **Golden Rule - Test data cannot influence the training phase at all**
- **No Free Lunch theorem - nothing works the best on every dataset etc etc.**

Naive Bayes

Bayes Rule

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)} = \frac{p(B|A)p(A)}{p(B) = \sum_{j=1}^{m} p(B|A_j)p(B_j)}$$

Because we want to know if $p(y_i = "spam"|x_i) > p(y_i = "notspam"|x_i)$, This lets us drop the denominator

$$\frac{p(x_i|y_i = "spam")p(y_i = "spam")}{p(x_i)} > \frac{p(x_i|y_i = "notspam")p(y_i = "notspam")}{p(x_i)}$$
$$p(x_i|y_i = "spam")p(y_i = "spam") > p(x_i|y_i = "notspam")p(y_i = "notspam")$$

- **For Naive Bayes we assume each variables in x_i is independent of the others in x_i given y_i**
- works well with tons of features compared to number of objects
- needs to know the probability of the features given the class
- Naive bayes model assumption showed in line 2-3 - all the features are conditionally independent given their label (y_i) - this is not true but a good approximation

$$\begin{aligned} p(spam|hello, vicodin, CPSC340) &= \frac{p(hello, vicodin, CPSC340|spam)p(spam)}{p(hello, vicodin, CPSC340)} \\ &\propto p(hello, vicodin, CPSC340|spam)p(spam) \\ &\approx p(hello|spam)p(vicodin|spam)p(CPSC340|spam)p(spam) \end{aligned}$$

```
k = max( y ) ;
counts = zeros ( k , 1 ) ;
for c = 1 : k
    counts(c) = sum( y==c ) ;
end

p_y = counts/n ; % This is the probability of each class , p( y(i) = c )
% We will store :
```

```
% p(x(i,j) = 1 | y(i) = c) as p_xy(j,1,c)
% p(x(i,j) = 0 | y(i) = c) as p_xy(j,2,c)
p_xy = zeros (d,2,k);

for c = 1 : k
    for j = 1 : d
        p_xy(j,1,c) = sum(X(y==c,j)==1)/ counts(c) ;
        p_xy(j,2,c) = sum(X(y==c,j)==0)/ counts(c) ;
    end
end
```

Parametric vs Non-Parametric

- Parametric models: fixed number of parameters - size of 'model' is $O(n)$ in terms of 'n'
- Non-Parametric models: number of parameters grows with 'n': size of model depends on 'n' - model gets more complicated with more data

K-nearest neighbours

Classification for KNN

```
For every object x:
    find k training examples x_i that are most similar to x
    classify using their mode
```

Condensed Version

```
inital subset is the first k examples
go through examples in order
if incorrectly classified by KNN, add example to training set
if correctly classified do not add it to the training set
```

KNN details

- non-parametric (grows with data)
- assumes objects with similar features have similar labels
- no training phase (lazy learning)
- define nearest via l1 norm - absolute, l2 norm - euclidean distance, etc.
- with a fixed k it has consistency properties
- with binary labels and mild assumptions - as n goes to infinity KNN test error is less than twice the irreducible error
- if k/n goes to zero and k goes to infinity:
- KNN is universally consistent = test error converges to irreducible error

Big O - $O(nd)$ to classify one test object (compute the similarities)

Random forests - average a set of deep decision trees, with bootstrap aggregation and random trees

Bootstrapping (bagging)

- Take a set of n objects chosen independently with replacement
- Generate several bootstrap samples of the objects (x_i, y_i), fit a classifier to each sample - at test time, average the predictions

Random trees

- When fitting each decision stump to construct deep decision tree
- do not consider all the features, each split looks at small number of randomly chosen features
- Each tree will tend to be very different from each other: they will still overfit but in different ways

```
function [model] = decisionForest(X,y,depth,nBootstraps)
[n,d] = size(X);
% Fit model to each bootstrap sample of data
for m = 1:nBootstraps
    randoms = randi(n,n,1);
    model.subModel{m} = randomTree(X(randoms,:),y(randoms),depth);
end
model.predict = @predict;
end

function [y] = predict(model,X)
% Predict using each model
for m = 1:length(model.subModel)
    y(:,m) = model.subModel{m}.predict(model.subModel{m},X);
end
% Take the most common label
y = mode(y,2);
end
```

Effect of number of trees/random-features

- When it comes to increasing the number of trees/submodels then the performance with bootstrapping converges to the performance without it

K-Means

Initialization

- Start with k initial means (usually random points), sensitive to outliers
- try several different random starting points, choose best one (minimizes average squared distance of data to means)

Error functions

- Functions that determine how 'right' we are and can improve the model (even to be convergent)
- L_2 norm -> euclidean distance -> bigger values are more important_
- L_1 norm -> absolute value -> all values are equal
- L_∞ norm -> max(r) -> only biggest value important

Big O - Total is $O(ndk)$

- Calculating distance from each x_i to each mean w_c is bottleneck - each time is $O(d)$ for all features
- For each n objects we compute the distance to k clusters
- Updating means is easier $O(nd)$ - sum objects in cluster divide by num objects in cluster

How to cluster new example

```
function [y] = predict(model,X)
[t,d] = size(X);
W = model.W;
k = size(W,1);

% Compute Euclidean distance between each data point and each mean
X2 = X.^2*ones(d,k);
distances = sqrt(X2 + ones(t,d)*(W').^2 - 2*X*W');

% Assign each data point to closest mean
[~,y] = min(distances,[],2);
end
```

Elbow method - choosing the k in which the the sharpest elbow (biggest change in slope) for minimum error vs k.

k-medians

- change objective function to L_1 norm (absolute value) from L_2 norm
- this is much more robust to outliers - update medians as median value of each cluster

Vector Quantization

- essentially replace vectors with a set of means

```
function [Iquant] = quantizeImage(I,b)
[n,m,d] = size(I);

% number of clusters can represent with a b-bit number is 2^b
k = 2^b;

% reshape the image to [n*m,3]
pixels = reshape(I,n*m,3);

model = clusterKmeans(pixels,k,0);
clusters = model.predict(model, pixels);

t = n*m;
for i = 1:t
   % replace each pixels color with the quantized one
   pixels(i,:) = model.W(clusters(i),:);
end

Iquant = reshape(pixels,n,m,d);
end
```

Density-based clustering

- clusters are defined by objects in dense regions
- non-parametric - clusters can become complicated the more data, no fixed # of clusters
- not sensitive to initialization (except boundaries)
- **Curse of dimensionality: problems with high-dimen spaces - volume of space grows exponentially with dimension**

Effect of parameters

- radius - minimum distance between points considered close or reachable
- minpoints - # of reachable points needed to define a cluster, if you have enough considered core
- merge core points if reachable from each other
- Sometimes may miss overarching or hierarchical clusters so we may need to play with parameters
- fix minpoints, record clusters as you vary radius, much more information than using a fixed radius

Shape of clusters

- can find non-convex clusters unlike kmeans/medians, doesn't cluster all the points

Big O

- computing distance between two examples is $O(d)$
- n training examples so $O(n^2)$ examples
- Computing distance between all examples is $O(n^2 d)$

summation and vector/matrix/norm notation

$$\sum_{i=1}^{n} |w^T x_i - y_i| = ||Xw - y||_1$$

$$max_{\epsilon 1,2,\ldots,n} |w^T x_i - y_i| + \frac{\lambda}{2} \sum_{j=1}^{n} |w_j| = ||Xw - y||_\infty + \frac{\lambda}{2} w^T w$$

$$\sum_{i=1}^{n} z_i (w^T x_i - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j| = (Xw - y)^T Z (Xw - y) + \lambda ||w|| 1$$

- Absolute value -> L_1 norm
- Max -> L_∞ norm
- Distance (or $x^T x$) -> L_2 norm (distance squared -> L_2 norm squared)

minimizing quadratic functions as linear systems

$$\nabla[c] = 0$$
$$\nabla[w^T b] = b$$
$$\nabla[\frac{1}{2} w^T A w] = Aw$$

function	gradient
$f(w) = \frac{1}{2}\|\|\|w - v\|\|\|^2 = \frac{1}{2}((w-v)^T(w-v)) = \frac{1}{2}(w^T w - w^T v - v^T w + v^T v) = \frac{1}{2}(w^T w - 2w^T v + v^T v) = \frac{1}{2} w^T w - w^T v + \frac{1}{2} v^T v$	$\nabla f(w) = w - v = 0$, $w = v$
$f(w) = \frac{1}{2}\|\|\|w\|\|\|^2 + w^T X^T y = \frac{1}{2} w^T w + w^T X^T y$	$\nabla f(w) = w + X^T y = 0$, $w = X^T y$
$f(w) = \frac{1}{2}\|\|\|Xw - y\|\|\|^2 + \frac{1}{2} w^T A w = \frac{1}{2}(Xw-y)^T(Xw-y) + \frac{1}{2} w^T A w = \frac{1}{2}(w^T X^T X w - w^T X^T y - y^T X w + y^T y) + \frac{1}{2} w^T A w = \frac{1}{2}(w^T X^T X w - 2w^T X^T y + y^T y) + \frac{1}{2} w^T A w$	$\nabla f(w) = X^T X w - X^T y + Aw = 0$, $(X^T X + A)w = X^T y$, $w = (X^T X + A)^{-1} X^T y$
$f(w) = \frac{1}{2}\sum_{i=1}^{n} z_i (w^T x_i - y_i)^2 = \frac{1}{2}(Xw-y)^T Z (Xw-y) = \frac{1}{2}(Xw-y)^T (ZXw - Zy) = \frac{1}{2}(w^T X^T - y^T)(ZXw - Zy) = \frac{1}{2}(w^T X^T ZXw - w^T X^T Zy - y^T ZXw + y^T Zy)$	$\nabla f(w) = X^T ZXw - X^T Zy$, $0 = Z^T ZXw - X^T Zy$, $w = (Z^T ZX)^{-1} X^T Zy$

Linear regression - Least Squares Objective = $f(w) = \sum_{i=1}^{n} (wx_i - y_i)^2$

change of basis

- Allows us to fit a quadratic model by changing the features
- $y_i = w_0 + w^T x_i$ becomes $y_i = w_0 + w^T z_i$
- where x -> z, eg. $x = \begin{bmatrix}1\\2\end{bmatrix}$ -> $z = \begin{bmatrix}1 & 1 & (1)^2\\1 & 2 & (2)^2\end{bmatrix}$
- As polynomial degree increase, the training error goes down, but becomes a worse approximation of the test error - usual approach for selecting the degree = cross validation
- Can combine multiple bases to improve performance like linear/periodic
- Can fit better as n increases but eventually data doesn't help if basis isn't flexible enough
- Non-Parametric bases: size of basis (# of features) grows with 'n', model gets more complicated with more data

RBFs - Radial Basis Functions

- Non-Parametric bases that depend on distances to training points
- Most common is Gaussian RBF $g(\alpha) = exp(-\frac{\alpha^2}{2\sigma^2})$
- Variance (σ^2 - stdev squared) controls influence of nearby points
- Replace X: n by d matrix with Z: n by n (distances of all points to each other)

$$X = \begin{bmatrix}x_{1,1} & \dots & x_{1,d}\\ \vdots & \ddots & \vdots\\ x_{n,1} & \dots & x_{n,d}\end{bmatrix} \text{ becomes } Z = \begin{bmatrix}g(||x_1 - x_1||) & \dots & g(||x_1 - x_n||)\\ \vdots & \ddots & \vdots\\ g(||x_n - x_1||) & \dots & g(||x_n - x_n||)\end{bmatrix}$$

- To predict your Xtest becomes t by d, and Z becomes t by n, so your features is the number of training examples
- Can add bias and linear basis to Z as well

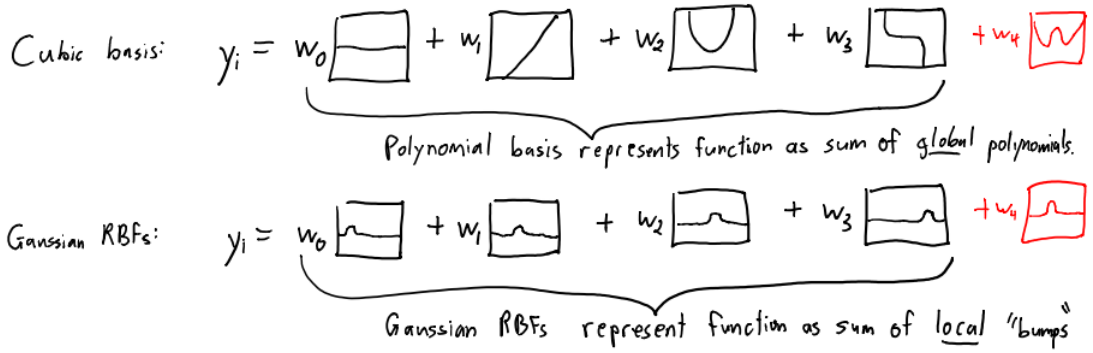

Figure 1:

regularization

- As we decrease variance we can make the training error go down
- the training error then becomes a worse approximation of the test error
- control this by using regularization: a penalty on complexity of the model $\frac{\lambda}{2} w^T w$
- large weight w_j leads to overfitting (cancel each other out)
- this gives objective function that minimize squared error + penalty on L2 norm of w
- balances getting a low error with the complexity of the model, reduces overfitting
- regularization parameter λ controls strength of regularization
  - as n grows λ should be in the range from $O(1)$ to $O(n^{1/2})$

cost of training/testing

- Linear Basis - $O(nd^2 + d^3)$
- $X^T X$ costs $O(nd^2)$ for $O(d^2)$ inner products at $O(n)$ each, and inverting $X^T X$ costs $O(d^3)$
- Classification costs $O(td)$ to compute t inner products at a cost of $O(d)$ each
- Gaussian RBF - $O(n^2 d + n^3)$
- Z costs $O(n^2 d)$ to compute $O(n^2)$ distances at a cost of $O(d)$ each.
- $Z^T Z$ costs $O(n^3)$ while inverting costs the same.
- Classification costs $O(tnd)$

effect of basis parameters

- if scales are different, penalizing the weight w_j means different things to different features
- can standardize features by subtracting its mean and dividing by its stddev.

Robust regression

weighted least squares - weighted by distance = locally linear regression

- or can use mean y_i among k-nearest neighbours
- put less focus on outliers
- use absolute error instead of squared error (L_1 norm instead of L_2^2 norm)
- setback -> cannot take gradient at 0 of absolute value function

smooth approximations

- harder to minimize non-smooth functions - so we can use approximations to absolute value
- Example approximation:
- Huber loss: squared error near zero (within some |r|), absolute value otherwise
- setting the gradient to zero does not give a linear system
- have to minimize f using gradient descent
- Gradient Descent -> starts with guess weight and uses that iteratively to find better one
- eventually the weight will converge of the gradient to zero, has some step size
- finds a global minimum on convex functions
- Normal equations cost $O(nd^2 + d^3)$ whereas GD costs $O(ndt)$ for t iterations
  - faster if solution is good enough for $t < d$ and $t < d^2$ / n
- In case we care about getting outliers correctly -> L_∞ norm (max), sensitive to outliers but better worst case
- convex but not smooth -> aka we can use GD
- log-sum-exp is a smooth approximation to max function
  - $max_i(z_i) \approx log(\sum_i exp(z_i))$
- Robust Regression using L1-norm/Huber is less sensitive to outliers
- Gradient descent finds local minimum of differentiable functions
- Convex functions do not have non-global local minima
- Log-Sum-Exp function: smooth approx to maximum
- function is convex iff $f''(w) \geq 0$ for all w
- norms and squared norms are convex, sum/max of convex functions is convex, composition of linear and convex is convex

computing gradients

- Usual inputs to a gradient method is a function that given w returns $f(w)$ and $\nabla f(w)$