6.867 Section 6: Nonparametric models

Contents

1 Intro **2**

2 Trees **2**
2.1 Regression . . . 3
2.1.1 Building a tree . . . 3
2.1.2 Pruning . . . 4
2.2 Classification . . . 4

3 Bagging **5**
3.1 Random Forests . . . 6

4 Boosting **7**
4.1 Adaboost Algorithm . . . 7
4.2 Additive models . . . 8
4.3 Margin . . . 10

5 Nearest-neighbor methods **10**
5.1 Regression . . . 11
5.2 Kernel density estimation . . . 12
5.3 Classification . . . 13
5.4 Curse of dimensionality . . . 13

1 Intro

We will continue to broaden the class of models that we can fit to our data. Graphical models allowed us to fit joint distributions that are compact to represent, and efficient for learning and inference. The complexity was adaptable, in the sense that structure search could try different dependence models and select one that was expected to be a good model of future data.

We now turn to models that are not directly interpretable as fitting a fixed-dimension distribution to data. The name *non-parametric methods* is misleading: it is really a class of methods that does not have a fixed parameterization in advance. Some non-parametric models, such as trees and boosting, which we might call *semi-parametric methods*, can be seen as dynamically constructing something that ends up looking like a more traditional parametric model, but where the actual training data affects exactly what the form of the model will be. Other non-parametric methods, such as nearest-neighbor and Bayesian non-parametric methods, rely directly on the data to make predictions and do not compute a model that summarizes the data.

The semi-parametric methods tend to have the form of a composition of simple models. We'll look at:

- *Tree models*: partition the input space and use different simple predictions on different regions of the space; this increases the hypothesis space.
- *Additive models*: train several different classifiers on the whole space and average the answers; this decreases the variance.

Boosting is a way to construct an additive model that both increases hypothesis space and decreases variance.

2 Trees

The idea here is that we would like to find a partition of the input space and then fit very simple models to predict the output in each piece. The partition is described using a (typically binary) "decision tree," which recursively splits the space.

These methods differ by:

- The class of possible ways to split the space at each node; these are generally linear splits, either aligned with the axes of the space, or more general.
- The class of predictors within the partitions; these are often simply constants, but may be probability distribution or more general classification or regression models.
- The way in which we control the complexity of the hypothesis: it would be within the capacity of these methods to have a separate partition for each individual training example.
- The algorithm for making the partitions and fitting the models.

The primary advantage of tree models is that they are understandable by humans. This is important in application domains, such as medicine, where there are human experts who think they know what they're doing and where decisions are critically important.

We'll concentrate on the CART/ID3 family of algorithms, which were invented independently in the statistics and the artificial intelligence communities. They work by greedily constructing a partition, where the splits are *axis aligned* and by fitting a *constant* model in the leaves. The interesting questions are how to select the splits and and how to control capacity. The regression and classification versions are very similar.

2.1 Regression

	No model	Prediction rule	Prob model	Dist over models
Regression		❋		

The classifier is made up of

- A partition function, π, mapping elements of the input space into exactly one of M regions, $R_1, \ldots, R_M$.
- A collection of M output values, O_m, one for each region.

If we already knew a division of the space into regions, we would set $\hat{y}_m$, the constant output for region R_m to be the average of the output values in that region; that is:

$$O_m = \text{average}_{\{i|x^{(i)} \in R_m\}} y^{(i)} \quad .$$

Define the error in a region as

$$E_m = \sum_{\{i|x^{(i)} \in R_m\}} (y^{(i)} - O_m)^2 \quad .$$

Ideally, we would select the partition to minimize

$$cM + \sum_{m=1}^{M} E_m \quad ,$$

for some regularization constant c. It is enough to search over all partitions of the training data (not all partitions of the input space) to optimize this, but the problem is NP-complete.

2.1.1 Building a tree

So, we'll be greedy. We establish a criterion, given a set of data, for finding the best single split of that data, and then apply it recursively to partition the space. We will select the partition of the data that *minimizes the sum of the mean squared errors of each partition.*

Given a data set D, let

- $R^+_{j,s}(D) = \{x \in D \mid x_j \geqslant s\}$
- $R^-_{j,s}(D) = \{x \in D \mid x_j < s\}$
- $\hat{y}^+_{j,s} = \text{average}_{\{i|x^{(i)} \in R^+_{j,s}(D)\}} y^{(i)}$
- $\hat{y}^-_{j,s} = \text{average}_{\{i|x^{(i)} \in R^-_{j,s}(D)\}} y^{(i)}$

BuildTree(D):

- If $|D| \leqslant k$: return Leaf(D)
- Find the variable j and split point s that minimizes:

$$E_{R^+_{j,s}(D)} + E_{R^+_{j,s}(D)} \quad .$$

- Return **Node**$(j, s,$ **BuildTree**$(R^+_{j,s}(D))$, **BuildTree**$(R^-_{j,s}(D))$

Each call to **BuildTree** considers $O(dn)$ splits (only need to split between each data point in each dimension); each requires $O(n)$ work.

2.1.2 Pruning

It might be tempting to regularize by stopping for a somewhat large k, or by stopping when splitting a node does not significantly decrease the error. One problem with short-sighted stopping criteria is that they might not see the value of a split that is, essentially, two-dimensional. So, we will tend to build a tree that is much too large, and then prune it back.

Define *cost complexity* of a tree T, where m ranges over its leaves as

$$C_\alpha(T) = \sum_{m=1}^{|T|} E_m(T) + \alpha|T| \ .$$

For a fixed α, find a T that (approximately) minimizes $C_\alpha(T)$ by "weakest-link" pruning. Create a sequence of trees by successively removing the bottom-level split that minimizes the increase in overall error, until the root is reached. Return the T in the sequence that minimizes the criterion.

Pick α using cross validation.

2.2 Classification

	No model	Prediction rule	Prob model	Dist over models
Classification		❋		

The strategy for building and pruning classification trees is very similar to the one for regression trees.

The output is now the majority of the output values in the leaf:

$$O_m = \text{majority}_{\{i|x^{(i)} \in R_m\}} y^{(i)} \ .$$

Define the error in a region as the number of data points that do not have the value O_m:

$$E_m = \left|\{i \mid x^{(i)} \in R_m \text{ and } y^{(i)} \neq O_m\}\right| \ .$$

Define the *empirical probability* of an item from class k occurring in region m as:

$$\hat{P}_{mk} = \hat{P}(R_m)(k) = \frac{\left|\{i \mid x^{(i)} \in R_m \text{ and } y^{(i)} = k\}\right|}{N_m}$$

We'll define the empirical probabilities of feature values, as well, for later use.

$$\hat{P}_{mjv} = \hat{P}(R_{mj})(v) = \frac{\left|\{i \mid x^{(i)} \in R_m \text{ and } x_j^{(i)} = v\}\right|}{N_m}$$

Splitting criteria Minimize "impurity" in child nodes. Some measures include:

- *Misclassification error*:
$$Q_m(T) = \frac{E_m}{N_m} = 1 - \hat{P}_{mO_m}$$
- *Gini index*:
$$Q_m(T) = \sum_k \hat{P}_{mk}(1 - \hat{P}_{mk})$$

- *Entropy*:

$$Q_m(T) = H(R_m) = -\sum_k \hat{P}_{mk} \log \hat{P}_{mk}$$

  Choosing the split that minimizes the entropy of the children is equivalent to maximize the *information gain* of the test $X_j = v$, defined by

$$\text{infoGain}(X_j = v, R_m) \quad = \quad H(R_m) - \left(\hat{P}_{mjv} H(R^+_{j,v}) + (1 - \hat{P}_{mjv}) H(R^-_{j,v})\right)$$

  Consider the two-class case. All have value

$$\begin{cases} 0.0 & \text{when } \hat{P}_{m0} = 0.0 \\ 0.5 & \text{when } \hat{P}_{m0} = 0.5 \\ 0.0 & \text{when } \hat{P}_{m0} = 1.0 \end{cases}$$

There used to be endless haggling about which one to use. It seems to be traditional to use:

- Entropy to select which node to split while growing the tree
- Misclassification error in the pruning criterion

Points about trees There are many variations on this theme:

- Linear regression or other regression or classification method in each leaf
- Non-axis-parallel splits: e.g., run a perceptron for a while to get a split.

What's good about trees:

- People understand them
- Easy to handle multi-class classification
- Easy to handle different loss functions (just change predictor in the leaves)

What's bad about trees:

- High variance: small changes in the data can result in very big changes in the hypothesis.
- Usually not the best predictions

Hierarchical mixture of experts Make a "soft" version of trees, in which the splits are probabilistic (so every point has some degree of membership in every leaf). Can be trained with a form of EM.

3 Bagging

Bootstrap aggregation is a technique for reducing the variance of a non-linear predictor, or one that is adaptive to the data.

- Construct B new data sets of size n by sampling them with replacement from $\mathcal{D}$
- Train a predictor on each one: $\hat{f}^b$

- *Regression case*: bagged predictor is

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^b(x)$$

- *Classification case*: majority bagged predictor: let $\hat{f}^b(x)$ be a vector with a single 1 and $K-1$ zeros, so that $\hat{y}^b(x) = \arg\max_k \hat{f}^b(x)_k$. Then

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^b(x),$$

  which is a vector containing the proportion of classifiers that predicted each class k for input x; and the predicted output is

$$\hat{y}_{bag}(x) = \arg\max_k \hat{f}_{bag}(x)_k \quad .$$

  Alternatively, we can average the class probabilities from the individual classifier, which gives us an even lower-variance estimate.

In the case of regression and *squared error*, we can show that expected squared error of a classifier trained on a single data set of size n is bounded below by the squared error of a classifier that is an average over classifiers trained on infinitely many data sets of size n *drawn from the population*. This suggests that bagging (in which new training sets are drawn from the data set, not population) will also decrease error.

For classification under 0-1 loss, bagging can improve a good classifier, but it can also make a bad classifier worse. But, here's a way to understand its advantage:

From Dietterich, via Hastie, Tibshirani, and Friedman.

- Let the Bayes optimal decision at x be $Y(x) = 1$ in a two-class example.
- Suppose each "committee member" Y_b has an error rate $e_b < e < 0.5$
- Let $S_1(x) = \sum_{b=1}^{B} I(G_b(x) = 1)$ be the number of votes for class 1 given input x
- If the committee members are independent, $S_1(x) \sim \text{Bin}(B, 1-e)$ and so $\Pr(S_1 > B/2) \to 1$ as B gets large.

This is the "Wisdom of Crowds."

The main issue with this analysis is that it assumes the committee members are independent, which they definitely are not in this case.

Also, when we bag a model, any simple predictability is lost.

3.1 Random Forests

Random forests are collections of trees that are constructed to be de-correlated, so that using them to vote gives maximal advantage.
For $b = 1..B$

- Draw a bootstrap sample $\mathcal{D}_b$ of size n from $\mathcal{D}$
- Grow a tree on data $\mathcal{D}_b$ by recursively repeating these steps:
  - Select m variables at random from the d variables
  - Pick the best variable and split point among them
  - Split the node
- return tree T_b

Given the ensemble of trees, vote to make a prediction on a new x.

4 Boosting

We will explore a method for making an additive model, but where we explicitly construct new data sets for training each new member of the ensemble, so that new classifiers attempt to "make up for" weaknesses of the current committee.

Assume a two-class problem with $y \in \{-1, 1\}$. The *training error rate* of a predictor G is

$$\hat{E}(G) = \frac{1}{n} \sum_{i=1}^{N} I(y^{(i)} \neq G(x^{(i)})) \quad .$$

Expected error rate is

$$E_{X,Y} I(Y \neq G(X)) \quad .$$

G is a *weak classifier* if its expected error is less than 0.5.

Assume we have an algorithm WL that takes in (weighted) data sets $(x^{(i)}, y^{(i)}, w^{(i)})$ and produces a weak classifier that attempts to minimize *weighted training error rate*:

$$\hat{E}_W(G) = \frac{\sum_{i=1}^{n} w^{(i)} I(y^{(i)} \neq G(x^{(i)}))}{\sum_{i=1}^{n} w^{(i)}} = \frac{ww}{W} \quad .$$

I just took this opportunity to define ww as *weight on wrong predictions* and W as the total weight. We will also define *weight on correct predictions* as $wc = W - ww$.

4.1 Adaboost Algorithm

The boosting algorithm works in a loop: feeding the training data into WL, evaluating the resulting classifier G_1 on the data, reweighting it to place more emphasis that were classified incorrectly, feeding the reweighted data into WL to get G_2, etc. The final classifier has the form

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \quad .$$

Here is the *Adaboost.M1* algorithm:

1. Initialize data weights $w_1^{(i)} = 1/n$
2. For $m = 1 \ldots M$
   (a) $G_m = \mathcal{WL}(\mathcal{D}, w_m)$
   (b) Compute $E_{w_m}(G_m)$
   (c) Compute
   $$\alpha_m = \log \frac{1 - \hat{E}_{w_m}(G_m)}{\hat{E}_{w_m}(G_m)}$$
   (d) For all i,
   $$w_{m+1}^{(i)} = w_m^{(i)} \cdot \exp\left(\alpha_m I(y^{(i)} \neq G_m(x^{(i)}))\right)$$
   Some versions of boosting normalize the weights after this step.
3. Output
   $$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \quad .$$

Points that are misclassified in round m have their weights increased by $\exp(\alpha_m)$, increasing their relative influence in the next round. Another way to see the update is

$$w_{m+1}^{(i)} = w_m^{(i)} \cdot \begin{cases} \frac{1-\hat{E}_{w_m}(G_m)}{\hat{E}_{w_m}(G_m)} & \text{if } I(y^{(i)} \neq G_m(x^{(i)})) \\ 1 & \text{otherwise} \end{cases}$$

Our premise is that although $\mathcal{WL}$ may be stupid, it will be able to find a classifier with training error rate less than 0.5.

This view emphasizes that the $w^{(i)}$ remain positive and that, if the current classifier G_m is working very poorly (that is, that $\hat{E}_{w_m}(G_m)$ is near 0.5), then the weights of the points we got wrong is not increased by much. If, on the other hand, G_m is working very well, so α_m is high, then the examples it got wrong will have their weights increased very substantially.

Weak learners You can use any non-ridiculous classifier as a weak learner. A very popular choice is the class of "decision stumps": these are decision trees that just make a single split. It's a terrible classifier all by itself, but can be boosted into generating very good performance.

4.2 Additive models

We can see that boosting ends up fitting a classifier of the form

$$f(x) = \sum_{m=1}^{M} \beta_m b(x\,;\,\gamma_m) \ ,$$

where β_m are expansion coefficients and b represents a class of basis functions, parameterized by γ_m. We could try to fit this form directly to data but it is a very difficult optimization problem.

Forward stagewise additive modeling Instead, we will fit the model incrementally, or "stagewise". This will be computationally simpler and also end up helping guard against overfitting. The idea is to iteratively add new classifiers that greedily improve the loss of the overall classifier, but not to go back revisit the parameters or expansion coefficients of previous stages.
FSAM algorithm

1. Initialize $f_0(x) = 0$
2. For $m = 1 \ldots M$
   (a) Compute new component

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{n} L(y^{(i)}, f_{m-1}(x^{(i)}) + \beta b(x^{(i)}\,;\,\gamma))$$

   (b) Add it to the ensemble predictor

$$f_m(x) = f_{m-1}(x) + \beta_m b(x\,;\,\gamma_m) \ .$$

Return f_M

Adaboost as FSAM We can see Adaboost.M1 as an instance of FSAM with the exponential loss function

$$L(y, f(x)) = \exp(-yf(x)) \ .$$

At each stage of FSAM, it is necessary to solve the problem

$$\begin{aligned}(\beta_m, G_m) &= \arg\min_{\beta, G} \sum_{i=1}^{n} \exp\left(-y^{(i)}\left(f_{m-1}(x^{(i)}) + \beta G(x^{(i)})\right)\right) \\ &= \arg\min_{\beta, G} \sum_{i=1}^{n} w_m^{(i)} \exp\left(-y^{(i)}\beta G(x^{(i)})\right)\end{aligned}$$

where

$$w_m^{(i)} = \exp(-y^{(i)} f_{m-1}(x^{(i)})) \ .$$

For any positive β,

$$\begin{aligned}G_m &= \arg\min_{G} \quad \exp(-\beta) \sum_{y^{(i)} = G(x^{(1)})} w_m^{(i)} \ + \ \exp(\beta) \sum_{y^{(i)} \neq G(x^{(1)})} w_m^{(i)} \\ &= \arg\min_{G} \quad (\exp(\beta) - \exp(-\beta)) \sum_{i=1}^{n} w_m^{(i)} I(y^{(i)} \neq G(x^{(i)})) \ + \ \exp(-\beta) \sum_{i=1}^{n} w_m^{(i)} \\ &= \arg\min_{G} \sum_{i=1}^{n} w_m^{(i)} I(y^{(i)} \neq G(x^{(i)})) \\ &= \mathcal{WL}(\mathcal{D}, w_m)\end{aligned}$$

Using a negative β is like flipping the signs on the $y^{(i)}$, and just ask for the classifier to generate the negation of the correct answers. It doesn't add any value, so we will not consider doing so.

Now, given G_m, we can solve for β

$$\begin{aligned}\beta_m &= \arg\min_{\beta} \quad \exp(-\beta) \sum_{y^{(i)} = G_m(x^{(1)})} w_m^{(i)} \ + \ \exp(\beta) \sum_{y^{(i)} \neq G_m(x^{(1)})} w_m^{(i)} \\ \beta_m &= \arg\min_{\beta} \exp(-\beta) wc_m + \exp(\beta) ww_m \\ &= \arg\min_{\beta} \exp(-\beta)\ W_m(1 - \hat{E}_{w_m}(G_m)) \ + \ \exp(\beta)\ W_m \hat{E}_{w_m}(G_m)\end{aligned}$$

where $W_m = \sum_{i=1}^{n} w_m^{(i)}$. Taking the derivative with respect to β and setting to 0, we find that

$$\beta_m = \frac{1}{2} \log \frac{1 - \hat{E}_{w_m}(G_m)}{\hat{E}_{w_m}(G_m)} \ .$$

Now

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x) \ ,$$

so the next weights are

$$\begin{aligned}w_{m+1}^{(i)} &= \exp(-y^{(i)} f_m(x^{(i)})) \\ &= \exp(-y^{(i)}(f_{m-1}(x^{(i)}) + \beta_m G_m(x^{(i)}))) \\ &= \exp(-y^{(i)} f_{m-1}(x^{(i)})) \exp(-y^{(i)} \beta_m G_m(x^{(i)}))) \\ &= w_m^{(i)} \exp(-y^{(i)} \beta_m G_m(x^{(i)})) \\ &= w_m^{(i)} \exp(\beta_m (2I(G_m(x^{(i)}) \neq y^{(i)}) - 1)) \\ &= w_m^{(i)} \exp(2\beta_m I(G_m(x^{(i)}) \neq y^{(i)})) \exp(-\beta_m) \\ &= w_m^{(i)} \exp(\alpha_m I(G_m(x^{(i)}) \neq y^{(i)})) \exp(-\beta_m)\end{aligned}$$

This has the same effect as the adaboost reweighting, even though it multiplies by a fixed factor of $\exp(-\beta_m)$.

So! Adaboost minimizes exponential loss by doing stagewise minimization of weighted misclassification error. This behavior is very interesting. If we plot error vs iteration, we find that:

- Misclassification rate on the training set falls to zero and stays there
- Even after training error is 0, the exponential loss continues to decrease.

If we knew the true data distribution, exponential loss would be minimized by picking

$$f^*(x) = \arg\min_{f(x)} E_{Y|x}(\exp(-Yf(x))) = \frac{1}{2}\log\frac{\Pr(Y=1 \mid x)}{\Pr(Y=-1 \mid x)} \quad .$$

Somewhat sensitive to outliers (mis-labeled points). Can make it more robust using a different loss function (HTF likes binomial deviance), but then there isn't a closed form for the optimization so finding each G_m will require a gradient descent.

Better than bagging.

4.3 Margin

Define the *voting margin* of a point to be

$$\text{margin}_f(x,y) = \frac{yf(x)}{\sum_m |\alpha_m|} = \frac{y\sum_m \alpha_m G_m(x)}{\sum_m |\alpha_m|}$$

It is a number in $[-1,+1]$ and is positive iff f correctly classifies the example. It can be interpreted as a measure of confidence in the prediction and it continues to decrease with rounds of boosting.

Schapire et al showed that larger margins on the training set result in superior upper bounds on the generalization error. For any $\theta > 0$, with high probability,

$$E_{gen} \leqslant \hat{Pr}(\text{margin}_f(x,y) \leqslant \theta) + O\left(\sqrt{\frac{d}{n\theta^2}}\right) \quad ,$$

where $\hat{Pr}$ is the empirical probability of the event in the data set. Note that the bound is independent of M.

5 Nearest-neighbor methods

	No model	Prediction rule	Prob model	Dist over models
Regression	❋			
Classification	❋			
Density estimation	❋			

Nearest neighbor methods are some of the simplest and are often very effective. They make few model assumptions and require little or no "training time". They can be very expensive in space and/or time to make predictions.

The idea is just that we remember all the data we have ever seen, and don't attempt to try to compute a model of it. Given a query, we answer it "directly" using a computation on the data.

These methods are often, confusingly, called "kernel methods." The notion of kernel as a distance function is used here as well as in other "kernelized" learning algorithms, but the details of the algorithms are very different.

5.1 Regression

Remember all your data! $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}$.

k-nearest-neighbor (KNN) regression:

$$\hat{f}(x) = \frac{1}{k} \sum_{\{i | x^{(i)} \in N_k(x)\}} y^{(i)} \ ,$$

where *neighbors* $N_k(x)$ are the k points in $\mathcal{D}$ minimizing (typically) Euclidean distance $\|x - x^{(i)}\|$ from x.

Nadaraya-Watson Kernel-weighted average (KWA) : **KNN** is straightforward, but gives discontinuous (piecewise constant) regression. If we would prefer for it to be smooth, we can use

$$\hat{f}(x) = \frac{\sum_{i=1}^{n} K_\lambda(x, x^{(i)}) y^{(i)}}{\sum_{i=1}^{n} K_\lambda(x, x^{(i)})} \ ,$$

where the kernel is defined as

$$K_\lambda(x, x') = D\left(\frac{\|x - x'\|}{\lambda}\right) \ .$$

The distance function D can be

- Gaussian:
$$D(t) = \frac{1}{\sqrt{2\pi}} \exp(-t^2/2)$$
- Epanechnikov:
$$D(t) = \begin{cases} \frac{3}{4}(1 - t^2) & \text{if } |t| \leqslant 1 \\ 0 & \text{otherwise} \end{cases}$$

Many other choices of kernels. They have to be symmetric ($K(u) - K(-u)$) and integrate to 1.

Some properties of **KWA**:

- Fitted function is continuous
- λ is a *bandwidth* or *smoothing* parameter; larger λ implies lower variance (it is more smoothed) but higher bias
- **KWA** with a fixed window width keeps bias constant but variance depends on local data density.
- **KNN** keeps variance constant but bias depends on data density.

KNN adapts the neighborhood size to the data density, which is useful in places where, for example, data is sparse. We can unify **KNN** and **KWA** by seeing them as both using a kernel

$$K_\lambda(x, x') = D\left(\frac{\|x - x'\|}{h_\lambda(x)}\right) \ .$$

- For **KWA**, $h_\lambda(x) = \lambda$
- For **KNN**, $h_\lambda(x) = \|x - x_{[k]}\|$ where $x_{[k]}$ is the kth closest $x^{(i)}$ to x.

Locally weighted regression (LWR) : Locally weighted averages are biased at the boundaries. We can improve them by doing regression. Our predictor will use a different $\hat{\alpha}$ and $\hat{\beta}$ for each predicted point x:

$$\hat{f}(x) = \hat{\alpha}(x) + \hat{\beta}(x) \cdot x \ ,$$

where

$$\hat{\alpha}(x), \hat{\beta}(x) = \arg\min_{\alpha,\beta} \sum_{i=1}^{n} K_\lambda(x, x^{(i)}) \left[y^{(i)} - \alpha - \beta \cdot x^{(i)} \right]^2 \ .$$

Inside the sum, the K term is a local weight and the other term is a squared error. To minimize this, we let:

- X be the $n \times d+1$ data matrix with a column of 1's added
- $W(x)$ be an $n \times n$ diagonal weight matrix, with $W_{ii}(x) = K_\lambda(x^{(i)}, x)$.
- Y be the $n \times 1$ vector of $y^{(i)}$ values.

Then

$$\begin{aligned} \hat{f}(x) &= \begin{bmatrix} 1 \\ x \end{bmatrix} \left(X^T W(x) X\right)^{-1} X^T W(x) Y \\ &= L(x) Y \end{aligned}$$

Think of L as a weight vector combining the least-squares with the kernel weights. We call this the *equivalent kernel*.

We could try to remove bias in curved regions of f by using local polynomial regression...at risk of increased variance.

General points

- Select kernel width with cross validation. Leave-one-out cross validation is *really easy* for NN methods.
- Raw features may need to be *standardized* (scaled to have a fixed variance), or the Euclidean metric might not make any sense. Cylinders in a car compared to number of pounds weight.
- As number of dimensions increases it is hard to keep both bias and variance low (need a lot of data in a small neighborhood). More on this later.

5.2 Kernel density estimation

To estimate the density at a query point x, compute

$$\hat{P}(x) = \frac{\text{num}\{x^{(i)} \mid \|x^{(i)} - x\|^2 < \lambda\}}{n\lambda} \ .$$

Think of this as putting a rectangular bump of width λ and height $1/\lambda$ down, centered at each $x^{(i)}$. To get a density, divide by the number of bumps.

To make a smoother estimate, use a different kernel. This is known as a *Parzen density estimate*:

$$\hat{P}(x) = \frac{1}{n\lambda} \sum_{i=1}^{n} K_\lambda(x, x^{(i)}) \ .$$

5.3 Classification

Easy to do multiple classes:

$$\hat{f}(x) = \arg\max_{y} \text{ number of times } y \text{ occurs in } \{y^{(i)} \mid x^{(i)} \in N_k(x)\} \ .$$

Good for irregular decision boundaries. Pick k by LOOCV.
Theorem: The asymptotic error rate of 1NN is less than or equal to 2 times the Bayes error rate.

Bayes error rate is the error rate of the best possible predictor.

- Assume query point is equal to some training point (this will be true asymptotically).
- Bayes rule tells us to predict $y^* = \arg\max_y \Pr(y \mid x)$, which has error $1 - \Pr(y^* \mid x)$
- 1NN error is

$$\sum_{y=1}^{k} \Pr(y = k \mid x)(1 - \Pr(y = k \mid x) \ .$$

  The first term inside the sum is the probability we got label k for input x; the second is the probability of getting an error if we use it to make a prediction.

For two classes,

$$\begin{aligned} \text{1NN error} &= \Pr(Y = 1 \mid x)(1 - \Pr(Y = 1 \mid x)) \cdot 2 \\ &= (1 - \Pr(Y = 0 \mid x))(1 - \Pr(Y = 1 \mid x)) \cdot 2 \\ &\leqslant 2(1 - \Pr(Y = y^* \mid x)) \\ &\leqslant 2 \cdot \text{Bayes error rate} \end{aligned}$$

Useful because it can accommodate distance metrics that respect the underlying invariance properties in the data (e.g., rotation and translation of images).

5.4 Curse of dimensionality

In high dimensions, all the $x^{(i)}$ will be far from each other and x. Lots of cool examples.

If we want to find a sub-hypercube of a unit-dimensional hypercube in d dimensions that contains 10% of the volume, what would the length of the sides of the subcube be? If $n = 1$, then 0.1. If $n = 10$, then 0.8.

All sample points are close to the boundary:

- Let M data points be uniformly distributed in a d-dimensional unit hypersphere
- Make a query at the origin
- The median distance to the closest data point is

$$\left(1 - \frac{1}{2}^{1/M}\right)^{1/d} \ .$$

- For $M = 5000$, $d = 10$, this value is approximately 0.52!!
- So, if we're doing regression, we're mostly extrapolating.

Moral of the story: if your data are *uniformly* distributed in high dimensions, these methods are hopeless.

But! If they live in a low-dimensional subspace of that high-dimensional space, it might turn out okay.