Day 14

30 Days of Machine Learning

Boosting:

Train Next Learner on mistakes made by previous learner(s)

Prepared By

Ahmed Ali

Quaid-e-Awam University of Engineering Science & Technology Nawabshah

Boosting: Train Next Learner on mistakes made by previous Learner

In bagging, generating complementary base-learners is left to chance and to the un-stability of the learning method. In boosting, we actively try to generate complementary base-learners by training the next learner on the mistakes of the previous learners. The original boosting algorithm combines three weak learners to generate a strong learner. A weak learner has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a strong learner has arbitrarily small error probability.

Original Boosting Concept

Given a large training set, we randomly divide it into three. We use X1 and train d1. We then take X2 and feed it to d1. We take all instances misclassified by d1 and as many instances on which d1 is correct.

from X2, and these together form the training set of d2. We then take X3 and feed it to d1 and d2. The instances on which d1 and d2 disagree form the training set of d3. During testing, given an instance, we give it to d1 and d2; if they agree, that is the response, otherwise the response of d3 is taken as the output.

1. Split data X into {X1, X2, X3}
2. Train d1 on X1
   - Test d1 on X2
3. Train d2 on d1's mistakes on X2 (plus some right)
   - Test d1 and d2 on X3
4. Train d3 on disagreements between d1 and d2
   - Testing: apply d1 and d2; if disagree, use d3
   - Drawback: need large X

overall system has reduced error rate, and the error rate can arbitrarily be reduced by using such systems recursively, that is, a boosting system of three models used as dj in a higher system.
Though it is quite successful, the disadvantage of the original boosting method is that it requires a very large training sample. The sample should be divided into three and furthermore, the second and third classifiers are only trained on a subset on which the previous ones err. So unless one has a quite large training set, d2 and d3 will not have training sets of reasonable size.

AdaBoost:

Freund and Schapire (1996) proposed a variant, named AdaBoost, short for adaptive boosting, that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit. AdaBoost can also combine an arbitrary number of base learners, not three.

AdaBoost Algorithm:

Training:
For all $\{x^t, r^t\}_{t=1}^N \in \mathcal{X}$, initialize $p_1^t = 1/N$
For all base-learners $j = 1, \ldots, L$
Randomly draw $\mathcal{X}_j$ from $\mathcal{X}$ with probabilities p_j^t
Train d_j using $\mathcal{X}_j$
For each (x^t, r^t), calculate $y_j^t \leftarrow d_j(x^t)$
Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$
If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop
$\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$
For each (x^t, r^t), decrease probabilities if correct:
If $y_j^t = r^t$, then $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$
Normalize probabilities:
$Z_j \leftarrow \sum_t p_{j+1}^t$; $p_{j+1}^t \leftarrow p_{j+1}^t / Z_j$
Testing:
Given x, calculate $d_j(x), j = 1, \ldots, L$
Calculate class outputs, $i = 1, \ldots, K$:
$y_i = \sum_{j=1}^L \left(\log \frac{1}{\beta_j}\right) d_{ji}(x)$

The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p^t denotes the probability that the instance pair (xt, rt) is drawn to train the j^{th} base-learner.
Initially, all pt = 1/N. Then we add new base-learners as follows, starting from j = 1: Є denotes the error rate of dj.
AdaBoost requires that learners are weak, that is, Є j < 1/2,∀ j; if not, we stop adding new base- learners. Note that this error rate is not on the original problem but on the dataset used at step j. define βj = Є j, /(1 −Є j) < 1, and we set pt+1 = β_j p^t if d_j correctly classifies x^t ;Otherwise $p^t_{j+1} = p_j$ because $p^t_{j+1} = p^t_j$. Because p^t_{j+1} should be probabilities, there is a normalization where we divide p^t_{j+1} by t p^t_{j+1} , so that they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p^t_{j+1} with replacement, and is used to train d_{j+1}.
This has the effect that d_{j+1} focuses more on instances misclassified by d_j; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, decision stumps, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.

Stacking-Stack Generalization:

Stacked generalization is a technique proposed by Wolpert (1992) that extends voting in that the way the output of the base-learners is combined need not be linear but is learned through a combiner system, f (· |Φ), which is another learner, whose parameters Φ are also trained. (see the below given figure).

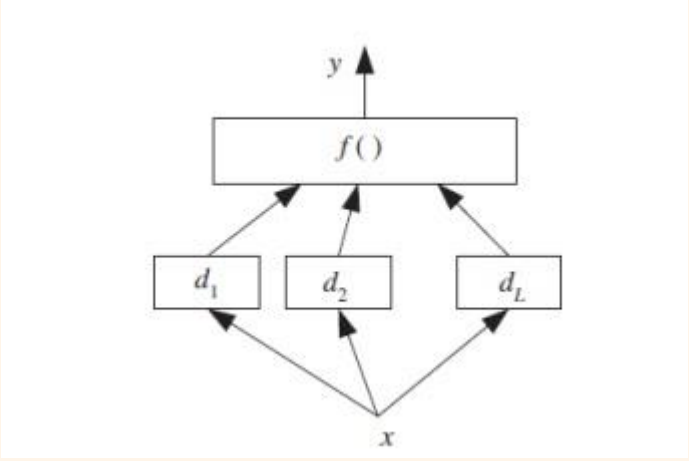


Figure: In stacked generalization, the combiner is another learner and is not restricted to being a linear combination as in voting.

$$y = f(d1, d2, \ldots, dL \mid \Phi)$$

The combiner learns what the correct output is when the base-learners give a certain output combination. We cannot train the combiner function on the training data because the base-learners may be memorizing the training set; the combiner system should learn how the base learners make errors. Stacking is a means of estimating and correcting for the biases of the base-learners. Therefore, the combiner should be trained on data unused in training the base-learners.
If $f(\cdot \mid w1, \ldots, wL)$ is a linear model with constraints, $wi \geq 0$, $\sum_j wj = 1$, the optimal weights can be found by constrained regression, but of course we do not need to enforce this; in stacking, there is no restriction on the combiner function and unlike voting, $f(\cdot)$ can be nonlinear. For example, it may be implemented as a multilayer perceptron with Φ its connection weights.
The outputs of the base-learners dj define a new L-dimensional space in which the output discriminant/regression function is learned by the combiner function.
In stacked generalization, we would like the base-learners to be as different as possible so that they will complement each other, and, for this, it is best if they are based on different learning algorithms. If we are combining classifiers that can generate continuous outputs, for example, posterior probabilities, it is better that they be the combined rather than hard decisions.
When we compare a trained combiner as we have in stacking, with a fixed rule such as in voting, we see that both have their advantages: A trained rule is more flexible and may have less bias, but adds extra parameters, risks introducing variance, and needs extra time and data for training.
Note also that there is no need to normalize classifier outputs before stacking.

Probabilistic Learning:

In machine learning, a probabilistic classifier is a classifier that can predict, given an observation of an input, a probability distribution over a set of classes, rather than only outputting the most likely class that the observation should belong to. Probabilistic classifiers provide classification that can be useful or when combining classifiers into ensembles.

One criticism that is often made of neural networks—especially the MLP—is that it is not clear exactly what it is doing: while we can go and have a look at the activations of the neurons and the weights, they don't tell us much.
In this topic (probabilistic classifier) we are going to look at methods that are based on statistics, and that are therefore more transparent, in that we can always extract and look at the probabilities and see what they are, rather than having to worry about weights that have no obvious meaning.

Gaussian Mixture Models:

However, suppose that we have the same data, but without target labels. This requires unsupervised learning, suppose that the different classes each come from their own Gaussian distribution. This is known as multi-modal data since there is one distribution (mode) for each different class. We can't fit one Gaussian to the data, because it doesn't look Gaussian overall.
There is, however, something we can do. If we know how many classes there are in the data, then we can try to estimate the parameters for that many Gaussians, all at once. If we don't know, then we can try different numbers and see which one works best. We will talk about this issue more for a different method (the k-means algorithm) in Unit 2. It is perfectly possible to use any other probability distribution instead of a Gaussian, but Gaussians are by far the most common choice. Then the output for any datapoint that is input to the algorithm will be the sum of the values expected by all the M Gaussians:

$$f(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m \phi(\mathbf{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m),$$

where _(x ; μm, m) is a Gaussian function with mean μm and covariance matrix m, and the αm are weights with the constraint that M m=1 α_m =1.
The given figures 1.1 shows two examples, where the data (shown by the histograms) comes from two different Gaussians, and the model is computed as a sum or mixture of the two Gaussians together.

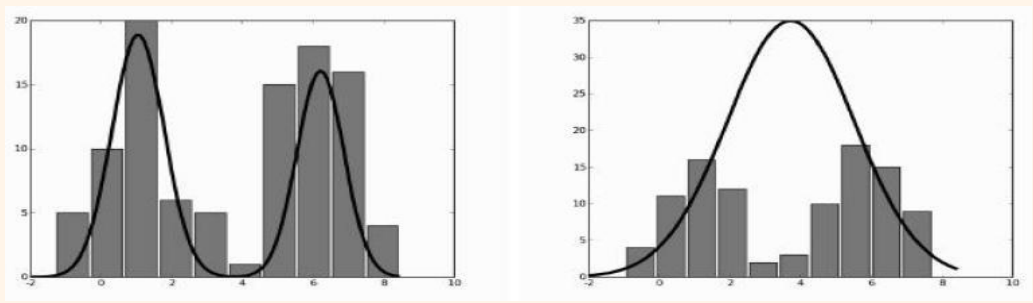


FIGURE 1.1: Histograms of training data from a mixture of two Gaussians and two fitted models, shown as the line plot. The model shown on the left fits well, but the one on the right produces two Gaussians right on top of each other that do not fit the data well.

The figure also gives you some idea of how to use the mixture model once it has been created. The probability that input xi belongs to class m can be written as (where a hat on a variable (ˆ ·) means that we are estimating the value of that variable):

$$p(\mathbf{x}_i \in c_m) = \frac{\hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_m; \hat{\boldsymbol{\Sigma}}_m)}{\sum_{k=1}^{M} \hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_k; \hat{\boldsymbol{\Sigma}}_k)}.$$

The problem is how to choose the weights α_m. The common approach is to aim for the maximum likelihood solution (the likelihood is the conditional probability of the data given the model, and the maximum likelihood solution varies the model to maximize this conditional probability). In fact, it is common to compute the log likelihood and then to maximize that; it is guaranteed to be negative, since probabilities are all less than 1, and the logarithm spreads out the values, making the optimization more effective. The algorithm that is used is an example of a very general one known as the expectation maximization (or more compactly, EM) algorithm.

Expectation-Maximization (EM) Algorithm:

The basic idea of the EM algorithm is that sometimes it is easier to add extra variables that are not actually known (called hidden or latent variables) and then to maximize the function over those variables. This might seem to be making a problem much more complicated than it needs to be, but it turns out for many problems that it makes finding the solution significantly easier.

To see how it works, we will consider the simplest interesting case of the Gaussian mixture model: a combination of just two Gaussian mixtures. The assumption now is that sample from that Gaussian. If the probability of picking Gaussian one is p, then the entire model looks like this (where N (μ, σ 2) specifies a Gaussian distribution with mean μ and standard deviation σ):

$$G_1 = \mathcal{N}(\mu_1, \sigma_1^2)$$
$$G_2 = \mathcal{N}(\mu_2, \sigma_2^2)$$
$$y = pG_1 + (1-p)G_2.$$

If the probability distribution of p is written as π, then the probability density is:

$$P(\mathbf{y}) = \pi\phi(\mathbf{y}; \boldsymbol{\mu}_1, \boldsymbol{\sigma}_1) + (1-\pi)\phi(\mathbf{y}; \boldsymbol{\mu}_2, \boldsymbol{\sigma}_2).$$

Finding the maximum likelihood solution (the maximum log likelihood) to this problem is then a case of computing the sum of the logarithm of Equation over all the training data, and differentiating it, which would be rather difficult. Fortunately, there is a way around it. The key insight that we need is that if we knew which of the two Gaussian components the datapoint came from, then the computation would be easy. The mean and standard deviation for each component could be computed from the datapoints that belong to that component, and there would not be a problem. Although we don't know which component each datapoint came from, we can pretend we do, by introducing a new variable f. If f = 0 then the data came from Gaussian one, if f = 1 then it came from Gaussian two.

This is the typical initial step of an EM algorithm: adding latent variables. Now we just need to work out how to optimize over them. This is the time when the reason for the algorithm being called expectation–maximization becomes clear.

We don't know much about variable f (hardly surprising, since we invented it), but we can compute its expectation (that is, the value that we 'expect' to see, which is the mean average) from the data:

$$\begin{aligned}\gamma_i(\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}) &= E(f|\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}, D) \\ &= P(f = 1|\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}, D),\end{aligned}$$

where D denotes the data. Note that since we have set f = 1 this means that we are choosing Gaussian two.
Computing the value of this expectation is known as the E-step. Then this estimate of the expectation is maximized over the model parameters (the parameters of the two Gaussians and the mixing parameter π), the M-step. This requires differentiating the expectation with respect to each of the model parameters. These two steps are simply iterated until the algorithm converges. Note that the estimate never gets any smaller, and it turns out that EM algorithms are guaranteed to reach a local maximum. To see how this looks for the two-component Gaussian mixture, we'll take a closer look at the algorithm:

The general algorithm has pretty much exactly the same steps (the parameters of the model are written as $\boldsymbol{\theta}$, $\boldsymbol{\theta}'$ is a dummy variable, D is the original dataset, and D' is the dataset with the latent variables included):

The General Expectation-Maximisation (EM) Algorithm

- Initialisation
  - guess parameters $\hat{\boldsymbol{\theta}}^{(0)}$
- Repeat until convergence:
  - (E-step) compute the expectation $Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}^{(j)}) = E(f(\boldsymbol{\theta}'; D')|D, \hat{\boldsymbol{\theta}}^{(j)})$
  - (M-step) estimate the new parameters $\hat{\boldsymbol{\theta}}^{(j+1)}$ as $\max_{\boldsymbol{\theta}'} Q(\boldsymbol{\theta}', \hat{\boldsymbol{\theta}}^{(j)})$

The trick with applying EM algorithms to problems is in identifying the correct latent variables to include, and then simply working through the steps. They are very powerful methods for a wide variety of statistical learning problems. We are now going to turn our attention to something much simpler, which is how we can use information about nearby datapoints to decide on classification output. For this we don't use a model of the data at all, but directly use the data that is available.

Information Criteria:

we introduced the idea of a validation set or using cross-validation if there was not enough data. However, this replaces data with computation time, as many models are trained on different datasets.
An alternative idea is to identify some measure that tells us about how well we can expect this trained model to perform. Probabilistic model selection (or "information criteria") provides an analytical technique for scoring and choosing among candidate models. Models are scored both on their performance on the training dataset and based on the complexity of the model.

There are two such information criteria that are commonly used:

Aikake Information Criterium

$$\mathrm{AIC} = \ln(\mathcal{L}) - k$$

Bayesian Information Criterium

$$\mathrm{BIC} = 2\ln(\mathcal{L}) - k\ln N$$

In these equations, k is the number of parameters in the model, N is the number of training examples, and L is the best (largest) likelihood of the model. In both cases, based on the way that they are written here, the model with the largest value is taken.

Nearest neighbor methods:

Suppose that you are in a nightclub and decide to dance. It is unlikely that you will know the dance moves for the song that is playing, so you will probably try to work out what to do by looking at what the people close to you are doing. The first thing you could do would be just to pick the person closest to you and copy them. However, since most of the people who are in the nightclub are also unlikely to know all the moves, you might decide to look at a few more people and do what most of them are doing. This is pretty much exactly the idea behind nearest neighbor methods: if we don't have a model that describes the data, then the best thing to do is to look at similar data and choose to be in the same class as them. We have the datapoints positioned within input space, so we just need to work out which of the training data are close to it. This requires computing the distance to each datapoint in the training set, which is relatively expensive: if we are in normal Euclidean space, then we must compute d subtractions and d squaring's (we can ignore the square root since we only want to know which points are the closest, not the actual distance) and this must be done $O(N^2)$ times. We can then identify the k nearest neighbors to the test point, and then set the class of the test point to be the most common one out of those for the nearest neighbors. The choice of k is not trivial. Make it too small and nearest neighbor methods are sensitive to noise, too large and the accuracy reduces as points that are too far away are considered. Some possible effects of changing the size of k on the decision boundary are shown in below Figure 1.2.

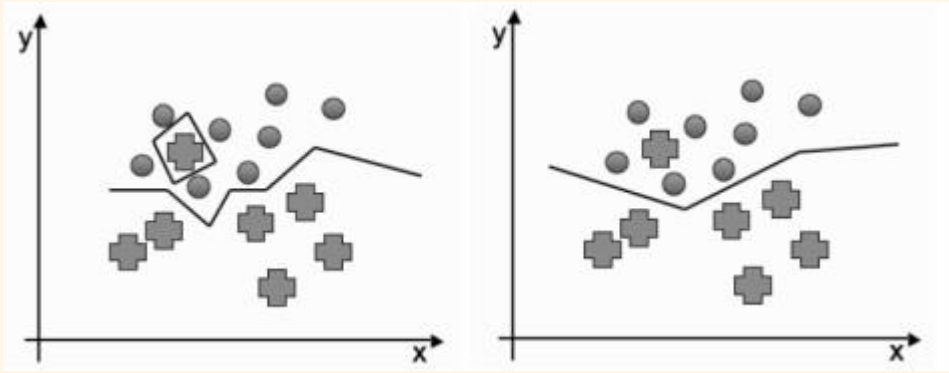


FIGURE 1.2: The nearest neighbor's decision boundary with left: one neighbor and right: two neighbors.

This method suffers from the curse of dimensionality. First, as shown above, the computational costs get higher as the number of dimensions grows. This is not as bad as it might appear at first: there are sets of methods such as KD-Trees (will discuss in upcoming topics) that compute this in O(N log N) time. However, more importantly, as the number of dimensions increases, so the distance to other datapoints tends to increase. In addition, they can be far away in a variety of different directions—there might be points that are relatively close in some dimensions, but a long way in others. There are methods for dealing with these problems, known as adaptive nearest neighbor methods, and there is a reference to them in the Further Reading section at the end of the chapter.

The only part of this that requires any care during the implementation is what to do when there is more than one class found in the closest points, but even with that the implementation is nice and simple:

```
def knn(k,data,dataClass,inputs):

    nInputs = np.shape(inputs)[0]
    closest = np.zeros(nInputs)

    for n in range(nInputs):
        # Compute distances
        distances = np.sum((data-inputs[n,:])**2,axis=1)

        # Identify the nearest neighbours
        indices = np.argsort(distances,axis=0)

        classes = np.unique(dataClass[indices[:k]])
        if len(classes)==1:
            closest[n] = np.unique(classes)
        else:
            counts = np.zeros(max(classes)+1)
            for i in range(k):
                counts[dataClass[indices[i]]] += 1
            closest[n] = np.max(counts)

    return closest
```

We are going to look next at how we can use these methods for regression, before we turn to the question of how to perform the distance calculations as efficiently as possible, something that is done simply but inefficiently in the code above. We will then consider briefly whether the Euclidean distance is always the most useful way to calculate distances, and what alternatives there are.

For the k-nearest neighbors' algorithm the bias-variance decomposition can be computed as:

$$E((\mathbf{y} - \hat{f}(\mathbf{x}))^2) = \sigma^2 + \left[f(\mathbf{x}) - \frac{1}{k} \sum_{i=0}^{k} f(\mathbf{x}_i) \right]^2 + \frac{\sigma^2}{k}.$$

The way to interpret this is that when k is small, so that there are few neighbors considered, the model has flexibility and can represent the underlying model well, but that it makes mistakes (has high variance) because there is relatively little data. As k increases, the variance decreases, but at the cost of less flexibility and so more bias.

Nearest Neighbor Smoothing:

Nearest neighbor methods can also be used for regression by returning the average value of the neighbors to a point, or a spline or similar fit as the new value. The most common methods are known as kernel smoothers, and they use a kernel (a weighting function between pairs of points) that decides how much emphasis (weight) to put onto the contribution from each datapoint according to its distance from the input.

Here we shall simply use two kernels that are used for smoothing. Both kernels are designed to give more weight to points that are closer to the current input, with the weights decreasing smoothly to zero as they pass out of the range of the current input, with the range specified by a parameter λ.

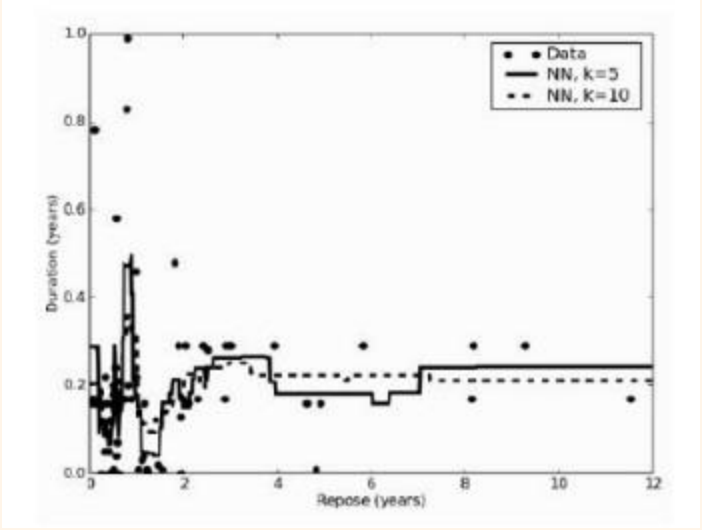
They are the Epanechnikov quadratic kernel:

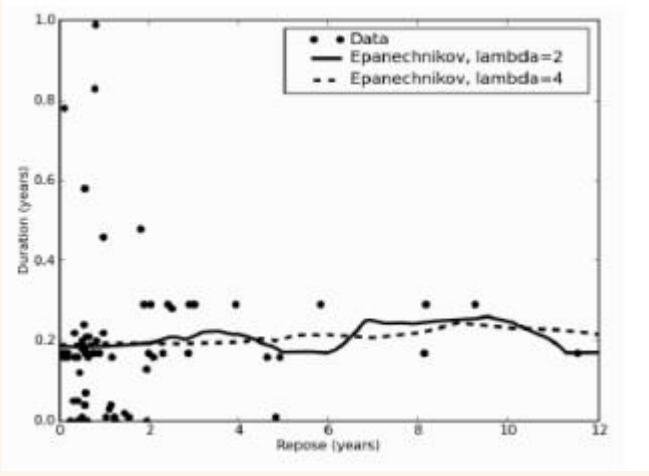
$$K_{T,\lambda}(x_0, x) = \begin{cases} \left(1 - \left|\frac{x_0 - x}{\lambda}\right|^3\right)^3 & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases},$$

and the tricube kernel:

$$K_{E,\lambda}(x_0, x) = \begin{cases} 0.75\left(1 - (x_0 - x)^2/\lambda^2\right) & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases},$$

The results of using these kernels are shown in below Figure 1.3 on a dataset that consists of the time between eruptions (technically known as the repose) and the duration of the eruptions of Mount Ruapehu, the large volcano in the center of New Zealand's north island. Values of λ of 2 and 4 were used here. Picking λ requires experimentation. Large values average over more datapoints, and therefore produce lower variance, but at the cost of higher bias.





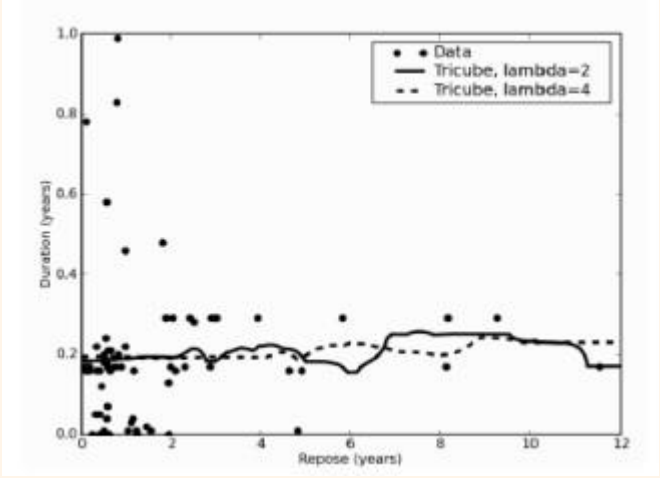


FIGURE 1.3: Output of the nearest neighbor method and two kernel smoothers on the data of duration and repose of eruptions of Mount Ruapehu 1860–2006.

Up-Next:

Efficient Distance Computations:

The KD-tree.