# 20 Noise in Data

If there was no noise in data, then data mining would be too easy, and a mainly solved topic.

What is noise in data? There are several main classes of noise, and modeling these can be as important as modeling the structure in data.

- **Spurious readings.** These are data points that could be anywhere, and are sometimes ridiculously far from where the real data should have been. With small data sets, these are often pruned by hand. With large sensed datasets, these need to be automatically dealt with. In high dimensions they are often impossible to physically "see."
- **Measurement error.** This is a small amount of error that can occur on *all* data points and in *every* dimension. It can occur because a sensor is not accurate enough, a rounding or approximation error occurs before the data reaches you, or a truncation error in a conversion to an abstract data type. This type of data noise may have little effect on the structure you are trying to measure, but can violate noiseless assumptions and should be understood.
- **Background data.** These data are from something other than what you are trying to measure. This could be unlabeled data (like unrated movies on Netflix) or people who don't have a disease you are trying to monitor. The problem is that it sometimes gets mixed in, and is then indistinguishable from the actual data of the phenomenon you are trying to monitor.

We will deal with this noise in 4 ways. First *cross validation* that helps one choose parameters, and also shows the influence of noise. Second is in identifying and maybe removing *outliers*, and is discussed later. Third is directly considering *uncertainty in data*, and modeling its effect on the output. Fourth is developing *robust estimators* to make the prediction impervious to a moderate amount of noise.

## 20.1 Cross Validation

Many algorithms we discuss have a parameter. **Probably all algorithms that deal effectively with noise have some parameter**. The $k$ in $k$-means clustering, the degree of a polynomial in polynomial linear regression, the number of singular values to use in PCA, the regularization parameter $s$ in Lasso. How should we choose this parameter?

We discussed many techniques. They key principal we emphasized was to reduce the problem to a simple and robust enough one that we could eye-ball it, e.g. the "elbow technique." Then we could hand choose the right parameter. If this seems un-satisfying, a more automated way is cross-validation.

The basic idea is simple: Divide $P$ (randomly) into two sets $Q, R \subset P$ so $Q \cap R = \emptyset$ and $Q \cup R = P$. Let $Q$ be the training set, and $R$ be the test set.

Then we run our algorithm to build a model on $Q$ with parameter $\gamma$: $M(Q, \gamma)$, and then evaluate the error $L_R(M(Q, \gamma))$ on $R$.

For instance, lets say the problem is to find a robust linear regression $A$ using Tikhinov Regularization minimizing the *loss function*:

$$L_P(A, \gamma) = \|P_y - P_X\alpha\|_2 + \gamma\|\alpha\|_2.$$

We solve $\alpha_{(Q,\gamma)} = (Q_X^T Q_X + \gamma^2)^{-1} Q_X^T Q_y$ and then evaluate $L_R(\alpha_{(Q,\gamma)}, 0)$. We can then search for the minimum value of $L_R(\alpha(Q, \gamma), 0)$ as a function of $\gamma$. This gives us a good choice of $\gamma$.

The idea behind this is that $P$ is drawn from some underlying distribution $\mu$, which we don't have access to. So we need to pretend we could draw more points from the true $\mu$. Instead of creating points out of the vapor, we use some points $R$ from $P$. But it is cheating to also train on these points so we need to keep the training set $Q$ and test set $R$ separate.

**Leave One Out Cross-Validation.** The above simple test set / training set problems gives too much emphasis to the arbitrary split of $R$ and $Q$. A less arbitrary way is to always let the test set $R$ be a single point $p$. This way we are essentially using as much data as possible to train $Q_p = P \setminus p$, and still getting a test. We can then repeat this for each point $p \in P$, and take the $\gamma$ which is best on average.

This can be too expensive (it requires $|P|$ constructions of our model). So sometimes we leave some $k$ (perhaps $= |P|/10$) points out each time and get $|P|/k$ different training / test set splits where each $p$ is tested once.

Or we can take out some $k$ number of test points at random and repeat this $t$ times, for a sufficiently large $t$.

*Testing on the Training Set (or the full set) can lead to* ***over-fitting*** where a model too complex is found, but seems to fit the data really well.

**Evaluating Generalization.** Sometimes the goal is not to optimize a parameter (say $k$), but to just evaluate how well a model fits the data. Again, doing this on the test data $R$ may give a biased answers, so one can train a model on $R$ and then test on $Q$.

In the same way, if one uses $R$ to train a model, and $Q$ to select a parameter (say $k$), then it is not fair to also evaluate the algorithm on $Q$. So in these cases, one should divide the initial set into three subsets $R$, $Q$, and $S$ so they are all disjoint. Then one can train on $R$, fit parameters on $Q$, and finally evaluate on $S$. This is now common place for many online contests (e.g. the Netflix challenge, or Kaggle); in these settings, often $R$ is given in raw, they provide a server that can evaluate algorithms (say once a day) on $Q$, and then at the end of the contest score all contestants using $S$.

It is important to note that even this three-way split can be abused. If one queries the test data $S$ too many times, then one can effectively learn it. Then it is also possible to overfit. In evaluating your experimental model, you should only use $S$ once. In Kaggle, it is often limited to once a day – but by creating multiple accounts, you can sometimes (illegally) get around this.

**Bootstrapping.** A downside of Leave-One-Out (LOO) cross validation (and even worse with its more efficient variants) is that it reduces the size of the test set $P$ artificially. If we have $|P|$ data points, we should be able to use all of them. Moreover with LOO (which only decreases to $|P| - 1$ points), we can only perform $|P|$ trials to average over. Perhaps we would like to do more to improve out estimate.

Bootstrapping (Efron 1979) chooses a set $Q \subset P$ **with replacement** of size $|P|$. So it has exactly the same size as $P$ itself. Then it trains on $Q$ to get $M(Q)$ and then also test on $Q$ as $L(M(Q), Q)$ to get an estimate of the error.

It then repeats this several times (maybe 1000 of 10,000 times) with new $Q$, each chosen independently with replacement from $P$.

This not only can be used for cross-validation, but is even more powerful (and designed for) understanding the effect of the noise in the data (with respect to how it is chosen from an imaginary $\mu$) on the model $M$.

For instance, one may find that the mean (as $M$) has much more variance than the median (as $M$) and be inclined to use the median instead of the mean.

## 20.2 Outliers

In general, *outliers are the cause of, and solution to, all of data mining's problems*. That is, any problem in data mining people have, they can try to blame it on outliers. And they can claim that if they were to remove the outliers the problem would go away. That is, when data is "nice" (there are no outliers) then the structure we are trying to find should be obvious. It turns out the solution to the outlier problem is at the same time simple and complicated.

Here we will discuss two general approaches for dealing with outliers:

- Find and remove outliers
- Density-based Approaches

### 20.2.1 Removal of Outliers

The basic technique to identify outliers is very straightforward, and in general should not be deviated from:

1. Build a model $M$ of dataset $P$
2. For each point $p \in P$ find the residual $r_p = \mathbf{d}(M(p), p)$, where $M(p)$ is the best representation of $p$ in the model $M$.
3. If $r_p$ is too large, then $p$ is an outlier.

Although this seems simple and fundamental, we needed to wait until this part of the class to define this properly.

- We have seen a variety of techniques for building models $M$ (clusters, regression).
- We have seen various distances that could make sense as $\mathbf{d}(M(p), p)$, and their trade-offs.
- We understand the difficulties of choosing a cut-off for what constitutes "too large."

It should also be clear that there is not one "right" way of deciding *any* of these steps. Should we do clustering or linear regression or polynomial regression? Should we look at vertical projection distance, use $L_1$ or $L_2$ distance? Should we remove the furthest $k$ points, or the furthest $3\%$ of points?

Consider a set $P$ that is drawn from a standard 1-dimensional Gaussian distribution. We could model this as a single cluster using the mean as a center. The the residual is the distance to the center. Now which points are outliers? Lets compare them against standard deviations?

- If we remove all points at $1$ standard deviation: we remove about 1/3 of all points.
- If we remove all points at $2$ standard deviations: we remove about 1/20 of all points.
- If we remove all points at $3$ standard deviations: we remove about 1/300 of all points.
- If we remove all points at $4$ standard deviations: we remove about 1/16000 of all points.

*So what is the right answer? ...*
Well, there is no "right" answer. If we fit the model with the right center, then why did we care about outliers in the first place; we still mined the right structure $M$! (Perhaps you say $M(p)$ should not be the mean, but some measure of how well it "fits" the Gaussian, but even in this approach you will reach a similar conclusion.)

*Another question is: does the process "converge"? ...*
If we remove the outliers from $P$, and then repeat, do we still find more outliers? Say we always remove the $10$ points with highest residuals. Then we can repeat this $|P|/10$ rounds until there are no points left. So you must have a way to determine if you should stop.

**Down-weighting.** One alternative to completely removing outliers is to just down-weight them. That is, when say finding the mean in $k$-means clustering, give inlier points a weight of $1$ (as normal) and outlier points a weight of say $1/2$ or $1/10$. This number may depend on the residual.

A more formal way of doing this can be through the use of a kernel (such as a Gaussian kernel), or some other similarity. Where points are weighted based on their similarity to the model $M$. Given a model $M$, points are reweighted, then the model is recomputed, and points are reweighted, and so on ...

### 20.2.2 Density-Based Approach

The idea is: (1) regular points have dense neighborhoods, and (2) outlier points have non-dense neighborhoods. Using the distance to the closest point is not robust, so we can use the distance to the $k$th closest point as measure of density. But what is $k$? Alternatively, we can measure density by counting points inside of a radius $r$ ball, or more robustly using the value of a kernel density estimate (using kernel with standard deviation $r$). But this requires a value $r$?

So this techniques needs a value $k$ or $r$, and then another threshold to determine what is in, and what is out. Sometimes the value $k$ or $r$ is apparent from the applications, and we can use an "elbow" technique for determining outliers.

But more seriously, this assumes that the density should be uniform throughout a data set. The "edge" will always have less density.

**Reverse Nearest Neighbors:** Given a point set $P$ for each point $p \in P$ we can calculate its nearest neighbor $q$ (or more robustly its $k$th nearest neighbor). Then for $q$ we can calculate its nearest neighbors $r$ in $P$ (or its $k$th nearest neighbor). Now if $\mathbf{d}(p,q) \approx \mathbf{d}(q,r)$ then $p$ is probably not an outlier.

This approach adapts to the density in different regions of the point set. It is also expensive to compute. And density (or relative density) may not tell you how far a point is from a model.

**Heavy-Tailed Distributions:** These types of distributions, which appear more and more frequently at internet scale, serve as a warning towards certain density filters for outliers.

Consider Zipf's Law: *the frequency of data is proportional to its rank.*

Let $X$ be a multiset so each $x \in X$ has $x = i \in [u]$. For instance all words in a book. Then $f_i = |\{x \in X \mid x = i\}|/|X|$. Now sort $f_i$ so $f_i \geq f_{i+1}$, then according to Zipf's law: $f_i \approx c(1/i)$ for some constant $c$.

For instance, in the Brown corpus, an enormous text corpus the three most common words are "the" at $7\%$ ($f_{\text{the}} = 0.07$); "of" at $3.5\%$ ($f_{\text{of}} = 0.035$); and "and" at $2.8\%$ ($f_{\text{and}} = 0.028$). So the constant is roughly $0.07$.

This also commonly happens when looking at customers at large internet companies. Amazon was built in some sense on the heavy tail.

- The top 10,000 books can be sold at physical book stores.
- The top 1 million books can be sold through Amazon, since they only need to sell say 1000 or less of each in the entire country to break even. They still stock these in their central warehouse.
- The top 100 million books can now still be sold by Amazon, but at a higher price; they can print them to order, or sell them on Kindle.

This is easy to see with books (and hence Amazon was built that way), but this happens with all sorts of internet products: movies, search results, advertising, food.

This has various screwy effects on a lot of the algorithms we study. And, to repeat, this rarely is seen at small scale, this only really comes into the picture in large rich data sets. Here are some of the screwy things that can happen:

- When doing SVD, there is no sharp drop in the singular values (which is *very* common with small datasets).
- If $30\%$ of words in a book occur less than $0.0001\%$ of the time (say occur at most twice), then can we just ignore them. Many LDA (text topic modeling softwares) do this, but should they really throw away $30\%$ of data. A subset of NLP-researchers think this puts an artificial cap on what LDA can do for text.

- We can find different structure at different scales, e.g. hierarchical PCA. Perhaps, each principal component is actually a series of 15 clusters (that happen to line up), and each of the clusters should have a component that points in a different direction. Think of classification of customers, or structure of human body (from atom, to protein, to cell, to organ, to being). Sometimes the components interact in ways that were not apparent at a large scale, so a pure hierarchy is never quite correct.

## 20.3 Missing Data (as Matrix Completion)

A common scenario is that a data set $P$ is provided, but is missing some (or many!) of its attributes. Lets focus on the case where $P \in \mathbb{R}^{n \times d}$ that is an $n \times d$ matrix, that we can perhaps think of as $n$ data points in $\mathbb{R}^d$, that is with $d$ attributes each. For instance, Netflix may want to recommend a subset of $n$ movies to $d$ customers. Each movie has only been seen, or even rated, by a small subset of all $d$ customers, so for each (movie, customer) pair $(i, j)$, it could be a rating (saying a score from $[0, 10]$), denoted $P_{i,j}$. But most pairs $(i, j)$ are empty, there is no rating yet. A recommendation would be based on the predicted score for unseen movies.

The typical notation defines a set $\Omega = \{(i,j) \mid P_{i,j} \neq \emptyset\}$ of the rated (movie, customer) pairs. Then let $\Pi_\Omega(P)$ describe the subset of pairs with scores and $\Pi_\Omega^\perp(P)$ the compliment, the subset of pairs without scores. The goal is to somehow fill in the values $\Pi_\Omega^\perp(P)$.

The simplest variants fill in the value as the average of all existing values in a row (the average rating in a movie). Or it could be the average of existing scores of a column (average rating of a customer). Or an average of these averages. But these approaches are not particularly helpful for personalizing the rating for a customer (e.g., a customer who likes horror movies but not rom-coms is likely to score things differently than one with the opposite preferences).

A common assumption in this area is that there are some simple ”latent factors” which determine a customers preferences (e.g., they like horror, and thrillers, but not rom-coms). A natural way to capture this is to assume there is some ”low-rank” structure in $P$. That is we would like to find a low-rank model for $P$ that fits the observed scores $\Pi_\Omega(P)$. The most common formulation looks like ridge regression:

$$P^* = \arg \min_{X \in \mathbb{R}^{n \times d}} \frac{1}{2} \|\Pi_\Omega(P - X)\|_F^2 + \lambda \|X\|_*.$$

Here $\|X\|_*$ is the *nuclear norm* of a matrix, it corresponds to the sum of its singular values (recall squared Frobenius norm is different since it is the sum of *squared* singular values), and it serves as a regularization term which biases the solution towards being low-rank.

A simple, and common way to approach this problems is iterative, and outlined in Algorithm 20.3.1. Start with some guess for $X$ (e.g., average of average of rows and of columns of $P_\Omega$). Take the svd of $X$ to obtain $USV^T \leftarrow \mathsf{svd}(X)$. Shrink all singular values by $\lambda$ or 0 (similar to frequent directions but not on the squared values). This operation $\phi_\lambda$ is defined for diagonal matrix $S$ as

$$\phi_\lambda(S) = \mathsf{diag}((S_{11} - \lambda)_+, (S_{22} - \lambda)_+, \ldots, (S_{dd} - \lambda)_+),$$

where $(x - \lambda)_+ = \max\{0, x - \lambda\}$. Then we update $\hat{X} = U\phi_\lambda(S)V^T$; this provides a lower rank estimate since $\phi_\lambda(S)$ will set some of the singular values to 0. Finally, we refill the known values $\Omega$ as

$$X \leftarrow \Pi_\Omega(P) + \Pi_\Omega^\perp(\hat{X}),$$

and repeat until things do not change much on an update step (it has “converged”).

This typically does not need too many iterations, but if $n$ and $d$ are large, then computing the SVD can be expensive. Various matrix sketching approaches can be used in its place to estimate a low-rank approximation more efficiently – especially those that pay attention to matrix sparsity. This is appropriate since the rank will be reduced in the $\phi_\lambda$ step regardless, and is the point of the modeling.

**Algorithm 20.3.1** Matrix-Completion($P$,$\Omega$,$\lambda$)

```
Initialize $X$ with each $X_{i,j}$ as
    $X_{i,j} = \begin{cases} \Pi_\Omega(P_{i,j}) & \text{if } (i,j) \in \Omega \\ \frac{1}{2}(\mathsf{average}_{i \in [n]}(P_\Omega(P_{i,j})) + \mathsf{average}_{j \in [d]}(P_\Omega(P_{i,j}))) & \text{otherwise} \end{cases}$
repeat
    $USV^T \leftarrow \mathsf{svd}(X)$
    $\hat{X} \leftarrow U\phi_\lambda(S)V^T$
    $X \leftarrow \Pi_\Omega(P) + \Pi_\Omega^\perp(\hat{X})$
until "converged"
return $\hat{X}$
```

## 20.4 Uncertain Data

Sometimes a data set $P$ is presented to a user with known uncertainty in it. Perhaps the way it was sensed indicates some distribution $\mu_i$ on the true value of each point $i \in P$. Or perhaps each object $p_i \in P$ is sensed multiple times so we have a representation $p_i = \{p_{i,1}, p_{i,2}, \ldots, p_{i,k}\}$. Thus each "point" $p \in P$ is actually represented as several (lets say $k$) different possible points. The common practice is then to treat $p_i$ as a probability distribution $\mu_i$ where

$$\mu_i : \mathbf{Pr}[p_i = x] = \begin{cases} 1/k & \text{if } x = p_{i,j} \in p_i \\ 0 & \text{otherwise.} \end{cases}$$

Lets be clear on the difference between this model and the cross-validation/bootstrapping model. Here we trust that each point exists, but it is drawn from an individual distribution $\mu_i$ (these are often assumed independent, but can have covariance). In the cross-validation model, we assume that *all* points are drawn from a single distribution $\mu$.

So how should we understand this uncertainty? Instead of a single model, we should have a *distribution of models*. This is most easily seen with a single point model, such as the mean, for one-dimensional data. In this can be seen as a distribution $f_P$ on the point that is the mean. In particular, let $f_P : \mathbb{R} \to \mathbb{R}^+$ represent the probability that its input is the mean. This is a bit hard to understand since when each $\mu_i$ is continuous (like a Gaussian) then all values $f_P(x) = 0$. It is easier to think of the cumulative density function $F_P(x) = \int_{z=-\infty}^{x} f_P(x)dx$.

Calculating $F_P$ exactly can be quite challenging, but it is easy to approximate. We say $g$ is an $\varepsilon$-*quantization* of $F_P$ if for all $x \in \mathbb{R}$ we have $|g(x) - F_P(x)| \leq \varepsilon$. We can create such a function $g$ as Algorithm 20.4.1. This generated an $\varepsilon$-quantization with probability at least $1 - \delta$; Since each $m_j$ is a random sample from $f_P$, we can apply a Chernoff-Hoeffding bound.

**Algorithm 20.4.1** Monte Carlo $(\varepsilon, \delta)$-Quantization

```
for $j = 1$ to $t = O((1/\varepsilon)^2 \log(1/\delta))$ do
    for all $p_i \in P$ do
        Choose some $q_{i,j} \sim \mu_i$.
    Let $Q_j = \{q_{1,j}, q_{2,j}, \ldots, q_{n,j}\}$.
    Calculate estimator $m_j$ from $Q_j$.
Let $\mathcal{M} = \{m_1, \ldots, m_t\}$.
Return $g(x) = |\{m_j \in \mathcal{M} \mid m_j \leq x\}|/t$.
```

This representation of $g$ is actually quite easy to work with since it is a step function defined by $t = O((1/\varepsilon)^2 \log(1/\delta))$ points. This value $t$ is *independent of $|P|$, the estimator (as long as it is a one-dimensional point), and all $\mu_i$!* However, $t$ can be quite large if $\varepsilon$ is small. For instance if $\varepsilon = 0.01$ (allowing $1\%$ error), and $\delta = 0.001$, then $t = 30{,}000$.

We can compress $g$ with the same error bounds by sorting the points $\mathcal{M}$ and taking only one out of every $t/s$ in the sorted order where $s = 1/\varepsilon$. Now the size is much smaller, where for $\varepsilon = 0.01$ only $s = 100$ points are needed.

A challenging open problem of the data is to extend this sort of approximate representation of uncertain output distributions for more complicated models and estimators.

## 20.5 Robust Estimators

*If you can't beat them, embrace them.*

The main problem with other approaches to handling noise/outliers is that in order to find a model $M$ to build residuals $\{r_p = \|p - M(p)\| \mid p \in P\}$ on determine outliers $\{p \in P \mid r_p > \tau\}$, is that it needs a good model $M$. But if we already have a good model $M$, then why do we can about outliers?

So here we discuss properties of techniques that build a model $M$ and are resistant, or *robust*, to outliers. Given a model $M(P)$, its *breakdown point* is an upper bound of the fraction of points in $P$ that can be moved to $\infty$ and for $M(P)$ not to also move infinitely far from where it started. For instance for single point estimators in the $1$ dimension, the mean has a break-down point of $1/n$, while the median has a breakdown point of $1/2$. An estimator with a large breakdown point is said to be a *robust estimator*.

In general, most estimators that minimize the sum of square errors (like PCA, least squares, $k$-means, and the mean) are *not* robust, and thus are susceptible to outliers.

Techniques that minimize the sum of errors (like least absolute differences, Theil-Sen estimators, $k$-median clustering, and the median) are robust, and are thus not as sensitive to outliers. Notice an analog to PCA is not there. This is an open research question of what the best answer is, as far as I know.

However, many of the $L_1$-regularization techniques (like Lasso) have the easy-to-solve aspects of least squares, but also simulate some of the robustness properties.