Interpretable Machine Learning: Multipurpose, Model-free, Math-free Fuzzy Regression

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.1, May 2022

Abstract

The innovative technique discussed here does much more than regression. It is useful in signal processing, in particular spatial filtering and smoothing. Initially designed using hyperplanes, the original version can be confused with support vector machines or support vector regression. However, the closest analogy is fuzzy regression. A weighted version based on splines makes it somewhat related to nearest neighbor or inverse distance interpolation, and highly non-linear. In the end, it is a kriging-like spatial regression, with many potential applications ranging from compression to signal enhancement and prediction. It comes with confidence intervals for the predicted values, despite the absence of statistical model. A predicted value is determined by hundreds or thousands of splines. The splines play the role of nodes in neural networks. Unlike neural networks, all the parameters – the distances to the splines – have a natural interpretation.

The methodology was tested on synthetic data. The performance, depending on hyperparameters and the number of splines, is measured on the validation set, not on the training set. Despite (by design) nearly perfect predictions for training set points, it is robust against outliers, numerically stable, and does not lead to overfitting. There is no regression coefficients, no intercept, no matrix algebra involved, no calculus, no statistics beyond empirical percentiles, and not even square roots. It is accessible to high school students. Despite the apparent simplicity, the technique is far from trivial. In its simplest form, the splines are similar to multivariate Lagrange interpolation polynomials. Python code is included in this document.

Contents

1 Introduction

The original problem consisted of fitting a line to a set of points – a classic linear regression problem. I explored alternatives to the traditional ordinary least squares (OLS) solution [Wiki]. The line that yields the least absolute residuals (LAR) [Wiki] is such an example. It has the benefit of being more robust. The next step was to look at all potential line combinations. For a set of n points, there are $M = n(n-1)/2$ potential lines, as each pair of points determines a line. The LAR line is just one of them and in some sense, the best one.

The idea is that for any local location x on the real axis, one can choose between multiple lines $z = L_k(x)$ to compute the predicted response z, with $k = 1, \ldots, M$. Some lines provide a better fit than others, at the local level. Or in other words, each of the M lines has some unique, location-dependent probability to contribute

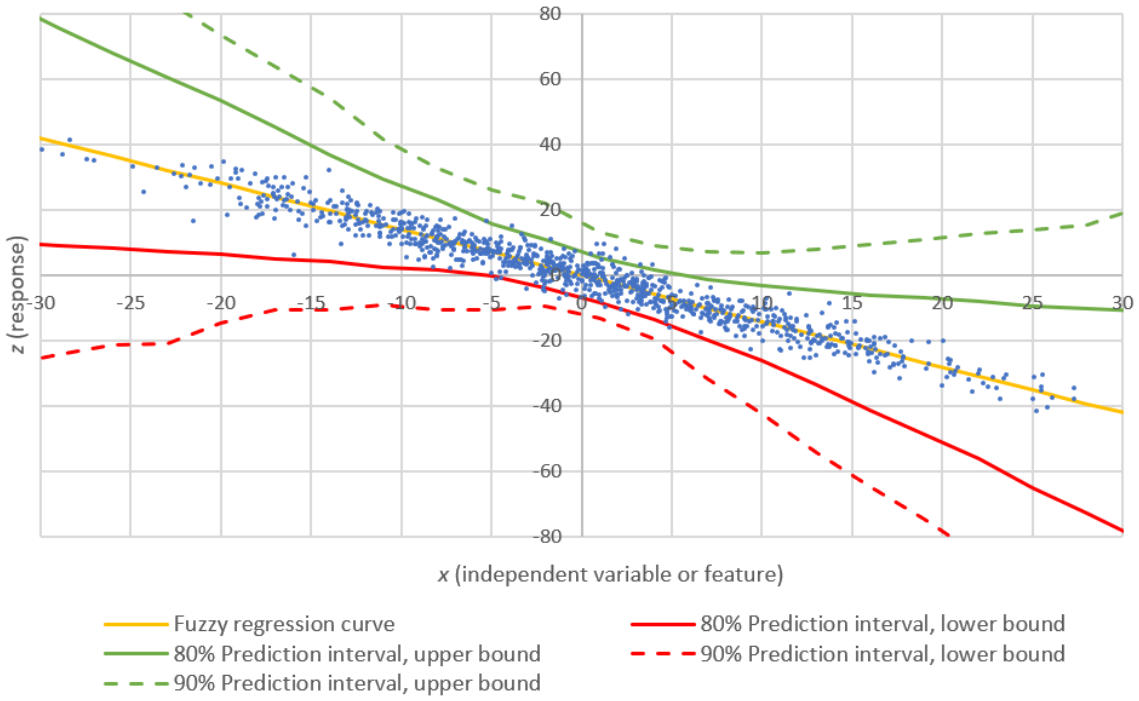

Figure 1: Fuzzy regression with prediction intervals, original version, 1D

to the predicted response computed at x. This perspective is very similar to the Bayesian approach. In the literature, this is known as the Theil-Sen estimator [Wiki]. In the simplest version, the median value of $L_k(x)$ computed across the M lines, is the final point estimate of the response, at location x. An improved version uses different weights for each line, involving weighted averages or weighted quantiles [1]. Since there are M potential predicted values attached to each x – one for each line – you can define a 80% confidence interval for the response, as follows: the lower (resp. upper) bound is the 10% (resp. 90%) empirical quantile [Wiki] (also called percentile) of the M predicted values computed at x.

The term prediction interval [Wiki], rather than confidence interval, is used in the literature. Note that the methodology to build these confidence intervals should not be confused with the percentile bootstrap method [Wiki]. Here, there is no resampling involved. The M lines are computed on the training set, while model performance is measured on the validation set [Wiki]. Note that I use the word "model" to represent the embodiment described in this article. There is no statistical or probabilistic model involved.

2 Original version

Before moving to the full, non-linear model in higher dimensions, let's focus on the original method: the first version of my fuzzy regression technique. This version is easier to understand, more traditional, and leads to simple visualizations. It will help you better understand the new version, which is considerably more abstract and generic.

Let the n observed points be labeled as $(x_1, z_1), \ldots, (x_n, z_n)$. The line that contains (x_i, z_i) and (x_j, z_j) is denoted as $L_k = L_{i,j}$. Its equation is

$$L_{i,j}(x) = \frac{z_i - z_j}{x_i - x_j} \cdot x + \frac{x_i z_j - x_j z_i}{x_i - x_j}.$$

It immediately follows that

$$L_{i,j}(x_i) = z_i, \quad L_{i,j}(x_j) = z_j.$$

Note that if $x_i = x_j$, the equation does not make sense. I will address this issue in the general case. I also use the notation $z_{i,j} = L_{i,j}(x)$. It represents the predicted value of z at location x, based on line $L_{i,j}$ solely. In its simplest form, the predicted value at x is the median of the $z_{i,j}$.

Figure 1 shows the result of this regression technique. To the naked eye, the regression curve is indistinguishable from the traditional regression line. It is also indistinguishable from a straight line, but it is actually a curve. The data set used here and pictured in Figure 1 (the blue dots) comes from my previous article on interpretable regression [2]. It is a synthetic data set with $n = 1,000$ observations.

The originality of the fuzzy regression procedure is that it allows you to compute prediction intervals without any underlying statistical model or bootstrap / resampling techniques. However, it requires the computation of $L_k(x)$ for $k = 1, \ldots, M$, for each sampled value of x. In higher dimensions, it is natural to replace the lines L_k by hyperplanes. However, this is not the path that I chose. Instead of hyperplanes, I used splines. The reason is that it leads to trivial computations and more flexibility. In particular, it does not involve matrix products or inversions. It does not involve matrices at all, nor calculus, thus my claim that the methodology is accessible to high school students.

3 Full, non-linear model in higher dimensions

I now discuss the general model. For the sake of simplicity, I focus on the 2-dimensional case: a response z, with two features x, y. It is an important case, with applications in geostatistics [Wiki]. The d-dimensional case is not more complicated, but the notations quickly become cumbersome. The line L_k is now replaced by a spline, also denoted as L_k. But this time, $k = (k_1, \ldots, k_r)$ is a vector, with $1 \leq k_i \leq n$ $(i = 1, \ldots, r)$. Just like a plane is uniquely determined by exactly 3 points, we want the spline L_k to be uniquely determined by exactly r points, in this case $(x_{k_1}, y_{k_1}, z_{k_1}), \ldots, (x_{k_r}, y_{k_r}, z_{k_r})$. That is, we want

$$z_{k_i} = L_k(x_{k_i}, y_{k_i}), \quad i = 1, \ldots, r.$$

There is very simple type of splines satisfying this property, namely

$$L_k(x, y) = \sum_{i=1}^{r} \frac{z_{k_i}}{2} \left[\prod_{j \neq i} \frac{\psi(x - x_{k_j})}{\psi(x_{k_i} - x_{k_j})} + \prod_{j \neq i} \frac{\psi(y - y_{k_j})}{\psi(y_{k_i} - y_{k_j})} \right], \tag{1}$$

where ψ is any real-valued function satisfying $\psi(0) = 0$. In practice, one can choose the identity function for ψ. The resulting splines are then similar to multivariate Lagrange polynomials of degree $r - 1$, used for interpolation [4]. I now discuss the various features, issues, capabilities, and potential implementations of this type of regression.

3.1 Geometric proximity, weights, and numerical stability

From now on, I assume that ψ is the identity function. A key concept is the proximity between a location (x, y) in the plane, and a spline. When predicting z, given (x, y), one has to compute $L_k(x, y)$ for each spline L_k. The number of splines quickly increases with r. For a specific location (x, y), some splines are more relevant than others. The following metric measures the proximity to L_k:

$$\delta_k(x, y) = \left(\prod_{i=1}^{r} \max(|x - x_{k_i}|, |y - y_{k_i}|) \right)^{1/r}. \tag{2}$$

It is the geometric mean of r Chebyshev distances [Wiki] in $\mathbb{R}^2$. In particular, it is zero if any of these distances is zero. This essential property can not be satisfied with the arithmetic mean, but it is with the geometric mean. The relevance or importance of L_k, relative to (x, y), is then defined as the weight

$$w_k(x, y) = \exp[-s \cdot \delta_k(x, y)], \tag{3}$$

where $s > 0$ is the smoothing parameter. It is maximum when $\delta_k(x, y) = 0$. On the other side, some splines are always problematic, or may not even exist. These splines are identified by the accuracy metric

$$\epsilon_k = \prod_{i=1}^{r} \prod_{j=i+1}^{r} |x_{k_i} - x_{k_j}| \cdot |y_{k_i} - y_{k_j}|. \tag{4}$$

In particular, if $\epsilon_k = 0$, the spline L_k is undefined. It is a good practice to reject or ignore splines with $\epsilon_k < 10^{-5}$. It increases the numeral stability of the system.

In addition, some splines may produce outlier predictions, depending on the location (x, y). Such abnormal predictions should be ignored to boost performance, when blending the M predicted values $L_k(x, y)$ – one per spline – to compute the final predicted value and prediction interval at (x, y). This final predicted value is the median or weighted average computed across all splines at (x, y), after rejecting undesirable splines or predictions. Outlier predictions are detected and rejected in the Python code, via the hyperparameter `zzdevratio`.

There is no need to be overly aggressive when penalizing and rejecting undesirable individual splines or predicted values. The median does a good job at filtering out non-robust measurements. The more aggressive,

the fewer splines used, resulting in lower statistical confidence. At the extreme, some locations may end up with no predicted value at all: for such locations, the variable `count` in the Python code is equal to zero, and the counter `missing` is incremented by one. This may be a good thing, or not.

3.2 Predicted values and prediction intervals

In section 2, I used the median predicted value computed across all $M = n(n-1)/2$ splines, as the final predicted value for the response z, at a specific location. Here n is the number of observations in the training set. Then I used the quantiles of these M values, computed at the same location, to build the prediction interval. The same applies to the general case, but now, $M = \binom{n}{r}$ is a binomial coefficient. Note that the actual number of splines will depend on the location (x, y), and is smaller than M if some splines are rejected. In the Python code, each call to the function `F` generates a new, random spline. The number M is pre-specified and is chosen to be large (> 500) but much smaller than $\binom{n}{r}$. Also, I mostly used $r = 2$.

An alternative to the median is to use a weighted average to compute a predicted value. For the weight attached to spline L_k, use formula (3). These weights and the whole system were designed to satisfy the following property. Let (x, y) be the location of a training set point. Then the predicted value at (x, y), using the weights in question with $s \to \infty$ and $M = n$ carefully chosen splines, is identical to the observed value. This is true for instance if the index k_1 in $k = (k_1, \ldots, k_r)$ covers all integer values between 1 and n, that is, all training set points. In that case, there is always at least one index vector k such that $\delta_k(x, y) = 0$, corresponding to a spline containing (x, y, z), with a weight equal to one, dwarfing all other weights. For a formal proof, see exercise 1.

Indeed, when s is large, the weighted methodology is similar to inverse distance weighting (Shepard's method) [Wiki], or nearest neighbor interpolation [Wiki]. The weighted version is implemented in the Python code. Another way to include neighboring data in the predictions is to only use local splines determined by training set points close to the target location (x, y). Finally, an efficient implementation still needs to be developed. The methodology can easily be implemented using a parallel computer architecture.

3.3 Illustration, with spreadsheet

See figure 1 for an illustration of the original method. Here I focus on the general method discussed in section 3, in two dimensions, and with non-linear splines. Figure 2 illustrates several aspects of this technique. Unlike in figure 1 (the one-dimensional case), it is difficult to show residual errors or prediction intervals for specific locations, because the locations (x, y) are now 2-dimensional. A workaround is to show a scatter plot of observed values z versus the predicted values z_{m}. These are the blue dots in figure 2. The notation z_{m} stands for the predicted value based on the median. The predicted value based on the weighted average is denoted as z_{w}, and not shown in the picture. The observed value z is also denoted as z_{obs}. The dashed blue line shows the quality of the fit between predicted and observed values. The R-squared is 0.8096.

However, the slope of the dashed line is only 0.4640. Maybe you expected it to be close to 1: after all, a perfect fit means all the blue dots are on the main diagonal. In practice, the slope will always be between 0 and 1. The explanation is as follows. The regression technique (spatial regression, to be precise), acts as a smoother or noise filtering technique, damping amplitudes. To eliminate the damping effect, you need to restore the amplitude. This is easily done by standardizing the predictions, so that their mean and variance corresponds to that of the original signal (observed response) z measured on the training set. Doing so won't change the R-squared, as it is invariant under translation and multiplication. Indeed, the R-squared is the square of the correlation between z and z_{m}.

The prediction levels are based on the 20% (lower bound) and 80% (upper bound) empirical quantiles. Thus, the confidence level is 60%. It is not possible to directly show prediction bands on a scatterplot in any meaningful way. Instead, to each blue dot in figure 2, corresponds one green and one red dot: the upper and lower bounds of the prediction interval. For each blue dot, the associated red and green dots are all on a same vertical line (not shown), parallel to the vertical axis. The details are unimportant; in the end, figure 2 still gives a good sense of how the methodology performs, and how the prediction intervals look like.

The detailed implementation is in the Fuzzy4.xlsx spreadsheet. Most of the heavy computations are done in Python. The spreadsheet provides the final steps: prediction intervals and visualizations. It also includes the output file `fuzzy_big.txt` produced by Python. Now, I am going to discuss the various fields in that spreadsheeet.

3.3.1 Output fields

I focus on the `2-D` tab in the spreadsheet. It contains three separate sets of columns, organized as follows:

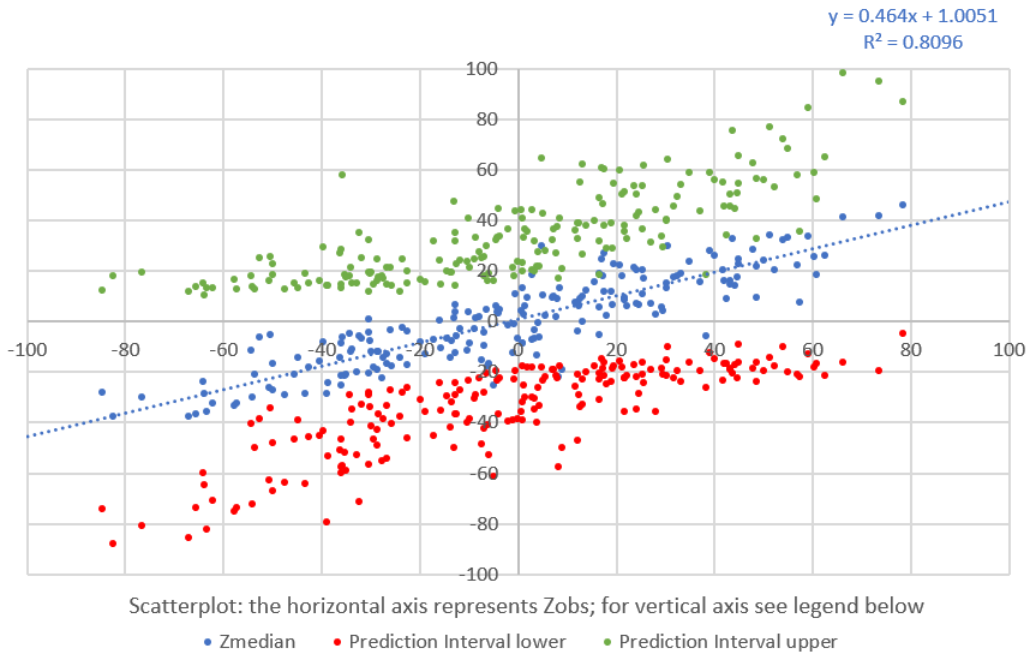

Figure 2: Fuzzy regression with prediction intervals, full model, 2D

- Columns `A`, `B`, `C`, `D` correspond respectively to x, y, z and the traditional predicted z using standard regression. It has 1000 rows, corresponding to the $n = 1000$ training set points. Only the first 800 points are used for training, the remaining 200 are used for validation.
- Columns `F` to `N` correspond to the output fields of `fuzzy_big.txt` produced by the Python code. It features the 200 points of the validation set, with for each point, up to $M = 800$ entries, one per spline. These are used to build the prediction intervals. Some splines are rejected as discussed in section 3.2, thus the actual number of rows is less than 800 per validation point. Columns `F - I` are trivial. Column `J` is the predicted value for the associated validation point in column `I`, arising from one single spline. The final predicted value for that point is the median of these values computed across all splines. It is stored in column `R`. Columns `K` and `L` correspond respectively to $\delta_k(x, y)$ and $w_k(x, y)$.
- Columns `P` to `U` correspond to summary statistics for each point of the validation set. Thus it has 200 rows, one per validation point. The median-based predicted value z_{m} is in column `R`, the weight-based predicted value z_{w} is in column `U`, the observed z is in column `Q`, and the lower and upper bounds of the prediction intervals are in columns `S` and `T`.

The cells `X2` and `Y2` in the `2-D` tab are the percentile levels for the prediction intervals. You can change these parameters, and it will automatically update figure 2 in the spreadsheet. The predicted z_{w}'s could be computed using the `AverageIf` Excel function. However there is no `MedianIf` or `PercentileIf` function in Excel. There is an easy workaround: for instance, instead of using the non-existent call `MedianIf(F:F,P2,J:J)`, use `Median(If(F:F=P2,J:J))`. This instruction means "compute the median value of column `J`, but only for rows that have the element in column `F` equal to cell `P2`". The same applies to the `Percentile` function.

4 Results

In this section, I present the main results. I tested the methodology on the synthetic data set described in my previous article [2]. It consists of $m = 1000$ observations with known response. The first 800 points are used to train the "model", and the remaining 200 – the validation data – for testing and assessing performance. The dataset is available in the spreadsheet fuzzyf2.xlsx, available on my GitHub repository. It corresponds to the small output file `fuzzy_small.txt` produced by the Python code in section 6. Predictions intervals are discussed in section 3.2 and illustrated in figure 2.

The main performance metric is the R-squared. It is certainly not the best metric for reasons discussed in my previous article [2], where I suggest alternatives. However, the dataset is large enough, and relatively well behaved. Thus the R-squared is adequate enough in this case. Note that it is mostly measured on the validation set rather than the training set, so technically it is not the true R-squared in the typical sense. Also, it is defined here as the square of the correlation coefficient. This is discussed in section 4.3.

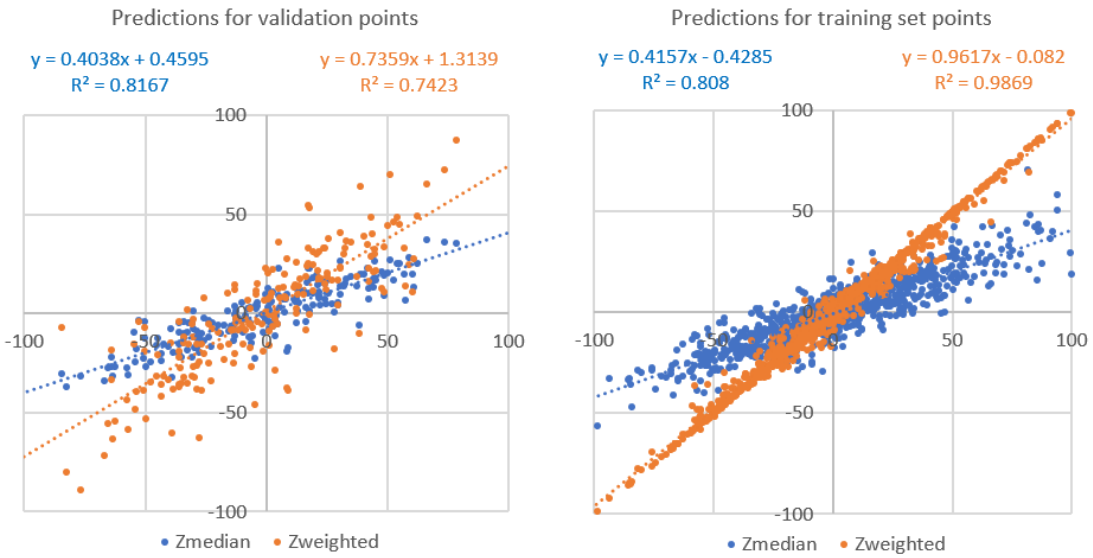

Figure 3: Scatterplots: median vs weighted method, on validation (left) vs training set (right)

4.1 Performance assessment

Table 1 summarizes my main experiment. The subscripts v, t, m, w stand respectively for the validation set, the training set, the median and the weighted predicted value. Regardless of the number M of random splines used per location, the weighted predicted value does best with splines defined by $r = 1$ point. Such splines have a constant value everywhere. This is not surprising, since this method, with $r = 1$, is similar to kriging or inverse distance interpolation. Note that with $r = 1$, the maximum number of distinct splines is $M = n$. Thus, if $n = 800$ and $M = 5000$, some splines are used multiple times.

The median predicted value is less sensitive to outliers, but it tends to reduce the amplitude, resulting in β values well below one. This is not an issue, as predicted values can easily be scaled back without impacting the R-squared. The median method, for this particular 2-D dataset, works best with $r = 2$. Also, it performs equally well inside and outside the training set. To the contrary, the weighted method experiences a sharp drop of performance, outside the training set. Larger values of r do not lead to further improvement. This is encouraging, as you want to work with small values of r and M to speed up of the algorithm. The larger r, the more potential splines to choose from, which leads to more accurate prediction intervals.

r	M	ρ^2_{vm}	ρ^2_{vw}	ρ^2_{tm}	ρ^2_{tw}	β_{vm}	β_{vw}	β_{tm}	β_{tw}
1	150	0.1922	0.6946	0.2490	0.7495	0.0916	0.7635	0.1066	0.7563
2	150	0.7688	0.4756	0.7525	0.7301	0.4128	0.6650	0.4210	0.7853
3	150	0.4272	0.3930	0.5601	0.5865	0.3032	0.6695	0.3306	0.7834
1	800	0.2600	0.7734	0.2936	0.8682	0.1025	0.7367	0.1039	0.8375
2	800	0.7941	0.6849	0.7913	0.9331	0.4058	0.7368	0.4154	0.9291
3	800	0.6838	0.5168	0.7204	0.9770	0.2986	0.6649	0.3392	0.9680
1	5000	0.2795	0.7876	0.4276	0.9294	0.1071	0.7421	0.1980	0.8945
2	5000	0.8167	0.7423	0.8080	0.9869	0.4038	0.7359	0.4157	0.9617
3	5000	0.7605	0.7203	0.7740	0.9988	0.3114	0.7063	0.3308	0.9892

Table 1: R-squared ρ^2 and slope β, on training and validation sets, median vs weighted

4.2 Visualization

Figures 2 and 3 further illustrate the methodology. The blue dots in the scatterplots represent the observed value (horizontal axis) versus the predicted value (vertical axis), computed using the median method. It provides a much better picture about the distribution of residual errors, than the R-squared alone. The orange dots show the same distribution of points, but computed using the weighted method instead. The fact that the

slopes are different is not an issue: the predicted values need to go though a final re-scaling step described in section 4.3, to correct the damping effect caused by the fuzzy regression, acting as a smoothing, low pass filter. Once corrected, the slopes will be nearly identical, and the R-squared unchanged.

Figure 2 is an original visualization, rarely seen. It allows you to look at individual residual errors and prediction intervals, regardless of the dimension of the problem.

4.3 Amplitude restoration

As mentioned a few times earlier, the fuzzy regression, especially the methodology based on the median, acts as a low-pass filter in signal processing. This is not surprising: after all it removes the noise. Indeed, it can be used as a data compression technique. As a result, predicted values have a lower variance than the observed ones, and the slope β in table 1 or figure 3 is well below one. To correct this "issue", one has to standardize the predicted values, so that the mean and variance match that of the observed response in the training set. In short, the predicted values must be re-calibrated. Because of this, the mean squared error is not a good metric to assess performance. Also, here the R-squared is the square of the correlation, but it is not equal to $1 - SS_{\text{res}}/SS_{\text{tot}}$, unlike in traditional regression where both agree.

The same phenomenon takes place when smoothing time series. A moving average can be used to predict or interpolate values. It also removes some noise, and reduces the amplitude of the signal. A scatterplot of exact values versus moving average will exhibit the same sharp drop in the slope. And it can be corrected using the same strategy, with no impact on the R-squared measured as the square of the correlation between observed and predicted (smoothed) values.

5 Exercises

The first exercise consists of proving a fundamental result: the fact that, under certain circumstances, the fuzzy regression technique described in this article is an exact interpolation technique. The proof does not involve math beyond elementary arithmetic, but rather, out-of-the-box thinking. The second exercise is about another simple, numerically stable interpolation technique, this time based on partial fractions. The prerequisite is a first course in calculus, to understand and solve this problem. The third exercise explores an alternative to validation sets. The fourth exercise is a generalization to higher dimensions.

Exercise 1 Fuzzy regression for interpolation. Let $(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)$ be the training set points. We use a spline system with $M = n$ splines. Each spline is uniquely defined by r training set points. The k-th spline ($k = 1, \ldots, n$) always contains (x_k, y_k, z_k). In other words, $L_k(x_k, y_k) = z_k$. Prove that as $s \to \infty$, the weight-based predicted value z_{w} evaluated at any training set location (x, y), is equal to the observed value z at that location.

Solution

When $s = \infty$, $w_k(x, y) = 1$ if $(x, y) = (x_k, y_k)$ is the location of a training set point, and 0 otherwise. If multiple training set points have the same (x, y) but different z's, then the predicted z_{w} at (x, y) will be the average of those z's. This is because at least one factor in the product formula (2) is equal to zero, and thus the product is zero.

To complete the proof, one has to carefully look at formula (1). Assume that $(x, y) = (x_k, y_k)$. If $i \neq k$, the i-th term in formula (1) is zero, because at least one factor in each inner product if zero. But if $i = k$, both products are equal to one, and thus $L_k(x, y) = z_k$.

Exercise 2 Partial fractions for interpolation. This may be particularly useful for time series interpolation. Assume f is a smooth, slow growing even function, and $f(t)$ is known if t is a positive integer. Then $f(t)$ is uniquely determined everywhere on the real axis. In short, there is an exact interpolation formula for the whole function, if we know $f(t)$ for $t = 0, 1, 2$ and so on. The formula is

$$f(t) = \frac{\sin \pi t}{\pi} \cdot \left[\frac{f(0)}{t} + \phi'(t) \sum_{k=1}^{\infty} (-1)^k \frac{f(k)}{\phi(t) - \phi(k)} \right], \tag{5}$$

and it works in particular if $\phi(t) = t^2$, $\phi'(t) = 2t$ is the derivative with respect to t, and

$$f(t) = \sum_{k=0}^{\infty} \alpha_k \cos \beta_k t, \text{ with } |\beta_k| < \pi. \tag{6}$$

The purpose of this exercise is to prove the validity of formula (5) under the right conditions, and to apply it to the real part of the Dirichlet eta function $\eta(t + i\sigma)$ [Wiki], for (say) $\sigma = 0.8$ and $0 < t < 30$. Unlike the

interpolation technique in exercise 1, formula (5) provides only an approximation, albeit an excellent one. The approximation is exact if you include all the infinitely many terms. It can be used for time series disaggregation [3]. A potential application is to break down hourly temperature predictions into 5 minute increments.

Solution

A detailed discussion about this interpolation formula and its generalization, can be found here. Note that the real part of the Dirichlet eta function (closely linked to the Riemann Hypothesis) is

$$\Re[\zeta(t+i\sigma)] = \sum_{k=1}^{\infty}(-1)^{k+1}\frac{\cos(t\log k)}{k^{\sigma}}, \quad \sigma > 0.$$

Figure 4 shows how accurate the interpolation formula is, for this particular example. The full function was reconstructed, based on $f(k)$ computed at $k = 0, 1, \ldots, 249$. The horizontal axis represents t. Note that to estimate $f(t)$ beyond $t = 30$, more than 250 terms are needed in formula (5), to keep the error smaller than 3×10^{-4}. Interestingly, the interpolation formula seems to be working even though condition (6) is not satisfied. At integer arguments, the error is minimum in absolute value, and smaller than 10^{-6}.

For the imaginary part – an odd function – you can multiply it by $\sin \lambda t$ to turn it into an even function, then apply the same methodology to the transformed function to interpolate it, then divide back by $\sin \lambda t$. Here $\lambda \neq 0$ is an arbitrary constant.

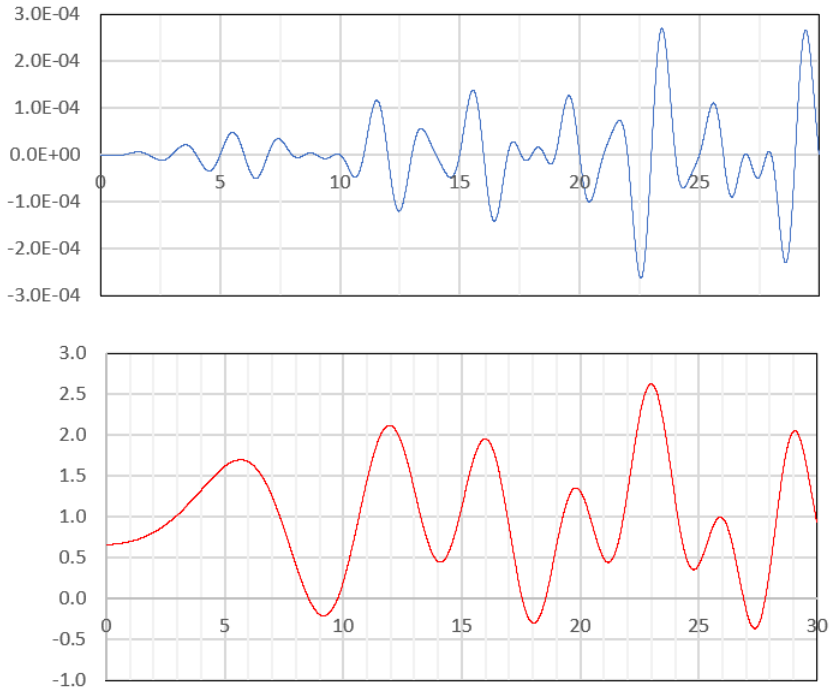

Figure 4: Dirichlet eta function (real part, bottom) and interpolation error (top)

Exercise 3 A new type of validation set. Since we are dealing with a regression problem, it is natural to see how the methodology performs on a linear combination of training set points. In other words, a validation point could be a convex linear combination [Wiki] of two or more training set points. A convex combination guarantees that the validation point is inside the convex hull of the training set points, and is good for interpolation. Also try with non-convex combinations, with validation points outside the convex hull. One would expect the performance to be lower for these points. It allows you to see how the method performs for extrapolation. Note that in d dimensions, the convex hull is obtained by computing all convex combinations of $d + 1$ points in the training set.

Exercise 4 Fuzzy regression in higher dimensions. In three dimensions, (x, y) becomes (x, x', x'') and formula (1) has three inner products. The weighted method will continue to work best with $r = 1$, but my guess is that the median method will work best with $r = 3$. The response is still denoted as z. In formula (2), $\max(|x - x_{k_i}|, |y - y_{k_i}|)$ becomes $\max(|x - x_{k_i}|, |x' - x'_{k_i}|, |x'' - x''_{k_i}|)$. Formula (4) is updated accordingly.

6 Python source code and datasets

I described the input/output data of the Python code in the previous sections. The Python source code is available on my GitHub repository, here. Below, it is broken down into four parts: commented introduction and setting the hyperparameter values, reading the input file, the core function, and the main part.

Part 1: the hyperparameters

```
# Kriging-style spatial regression / inverse distance interpolation

import numpy as np
import random
random.seed(100)

# --- Highlights of this "fuzzy regression" code:

# Model-free; produces big output file to compute prediction intervals; Bivariate case,
    featuring nearest-neighbor approach (the weights); Exact predictions for training
    set, yet robust (no overfitting); Increasing M is "lazy way" to boost performance,
    but it slows speed
# Math-free (no matrix algrebra, square root or calculus); Statistics-free (no
    statistical science involved at all); Requires no technical knowledge beyond high
    school, but far from trivial!
# Acts as low-pass, amplitude reduction, or signal compression filter; Also acts as
    noise filtering, signal enhancement. Amplitude restoration step not included, but
    easy to do.

# By Vincent Granville, www.MLTechniques.com

# --- Hyperparameters

# n (number of obs, called points) set after reading input file [n=1000 here]

P=0.8       # proportion of data allocated to training the remaining is for validation
M=5000      # max number of splines used per point; M=5000 offers modest gain over M=800
r=2         # number of points defining a spline; also works with r=1 or larger r
smoother=1.5 # smoothing param used in weighted predictions; try 0.5 for more smoothing
    (0 = max smoothing)
thresh1=25.0 # max distance allowed to nearby spline; increase to eliminate points with
    no predictions; decrease to narrow (improve) confidence intervals
thresh2=1.5 # max outlier level allowed for predicted values ; if < 1, predicted can't
    be more extreme than observed; if too low, may increase number of points with no
    prediction; if too large, may produce a few strong outlier predictions;
thresh3=0.001 # control numerical stability (keep 0.001)

# --- Output var (defined later)

# missing   : number of points not assigned a prediction
#
# count     : actual number of splines used for a specific point
# error     : code telling why a point is not assigned a prediction
# weight    : weight assigned to a spline, for a given point
# zpred     : predicted value for a point zz = (xx, yy)
# zpredw    : weighted predicted value

# Input var (defined later)
#
# xx, yy, zz: coordinates of a point
```

Part 2: reading the input file

```
# --- Reading input file

x=[]
```

```
y=[]
z=[]

file=open('fuzzy2b.txt',"r")
lines=file.readlines()
for aux in lines:
   x.append(aux.split('\t')[0])
   y.append(aux.split('\t')[1])
   z.append(aux.split('\t')[2])
file.close()

x = list(map(float, x))
y = list(map(float, y))
z = list(map(float, z))

zmin=np.min(z)
zmax=np.max(z)
zavg=np.mean(z)
zdev=max(abs(zmin-zavg),abs(zmax-zavg))

n=len(x)
```

Part 3: the core function

```
# --- Core function: spline-based interpolator

def F(xx,yy,r):

  zz=0
  distmin=1
  error=0

  idx=[]
  A=[]
  B=[]

  for i in range(0,r):
    idx.insert(i,int(n*P*random.random()))

  prod=1.0;
  for i in range(0,r):
    for j in range(i+1,r):
      prod*=(x[idx[i]]-x[idx[j]])*(y[idx[i]]-y[idx[j]])
  if abs(prod)>thresh3:
    for i in range(0,r):
      A.insert(i,1.0)
      B.insert(i,1.0)
      for j in range(0,r):
        if j != i:
          A[i]*=(xx-x[idx[j]])/(x[idx[i]]-x[idx[j]])
          B[i]*=(yy-y[idx[j]])/(y[idx[i]]-y[idx[j]])
      zz+=z[idx[i]]*(A[i]+B[i])/2
      distmin*=max(abs(xx-x[idx[i]]),abs(yy-y[idx[i]]))
    distmin=pow(distmin,1/r)
  else:
    error=1;

  return [zz,distmin,error]
```

Part 4: main step

```
# --- Main step: predictions for points in validation set
```

```
# For training set predictions, change range(int(P*n),n) to range(0,int(P*n))

file_small=open("fuzzy_small.txt","w")
file_big=open("fuzzy_big.txt","w")

for j in range(int(P*n),n): # loop over all validation points

  xx=x[j]
  yy=y[j]
  zobs=z[j]
  count=0
  missing=0
  sweight=0.0
  zpredw=0.0
  zpred=0.0

  for k in range(0,M): # inner loop over all splines

    list=F(xx,yy,r)
    zz=list[0]
    distmin=list[1]
    error=list[2]
    weight=np.exp(-smoother*distmin)
    zzdevratio=abs(zz-zavg)/zdev

    if distmin<thresh1 and zzdevratio<thresh2 and error==0:
      count+=1
      sweight+=weight
      zpredw+=zz*weight
      zpred+=zz
      row=[j,xx,yy,zobs,zz,distmin,weight,zzdevratio]
      for field in row:
        file_big.write(str(field)+"\t")
      file_big.write("\n")

  if count>0:
    zpredw=zpredw/sweight
    zpred=zpred/count
  else:
    missing+=1
    zpredw=""
    zpred=""

  row=[j,count,xx,yy,zobs,zpred,zpredw]
  for field in row:
    file_small.write(str(field)+"\t")
  file_small.write("\n")

file_big.close()
file_small.close()
print(missing,"ignored points\n")
```

References

[1] Weighted percentiles using numpy. *Forum discussion*, 2020. StackOverflow [Link]. 2

[2] Vincent Granville. Little known secrets about interpretable machine learning on synthetic data. *Preprint*, pages 1–14, 2022. MLTechniques.com [Link]. 2, 5

[3] Vincent Granville and Richard L Smith. Disaggregation of rainfall time series via Gibbs sampling. *NISS Technical Report*, pages 1–21, 1996. [Link]. 8

[4] Kamron Saniee. A simple expression for multivariate Lagrange interpolation. *SIAM Undergraduate Research Online*, 2007. SIURO [Link]. 3