Bootstrapping prediction intervals

Posted on March 1, 2020

Continuing from where we left off (https://saattrupdan.github.io/2020-02-26-parametric-prediction/), in this post I will discuss a general way of producing accurate prediction intervals for all machine learning models that are in use today. The algorithm for producing these intervals uses bootstrapping (https://en.wikipedia.org/wiki/Bootstrapping_%28statistics%29) and was introduced in Kumar and Srivastava (2012) (https://ntrs.nasa.gov/search.jsp?R=20130014367).

This post is part of my series on quantifying uncertainty:

1. Confidence intervals (https://saattrupdan.github.io/2020-02-20-confidence/)
2. Parametric prediction intervals (https://saattrupdan.github.io/2020-02-26-parametric-prediction/)
3. Bootstrap prediction intervals
4. Quantile regression (https://saattrupdan.github.io/2020-03-09-quantile-regression/)
5. Quantile regression forests (https://saattrupdan.github.io/2020-04-05-quantile-regression-forests/)
6. Doubt (https://saattrupdan.github.io/2021-04-04-doubt/)

The setup

To prove that the prediction intervals are valid the authors made some assumptions on both the true data distribution and our predictive model. Let's say that we're working with a d-dimensional feature space and that we only have a single response variable. We will then assume that the true model $y\colon \mathbb{R}^d \to \mathbb{R}$ is of the form

$$y(x) = \psi(x) + \varepsilon(x),$$

where $\psi\colon \mathbb{R}^d \to \mathbb{R}$ is the "main model function" and $\varepsilon\colon \mathbb{R}^d \to \mathbb{R}$ is a noise function. These will satisfy that

1. ψ should be **deterministic**, meaning that is has no random elements;
2. ψ is "sufficiently smooth";
3. $\varepsilon(x)$ are iid for all $x \in \mathbb{R}^d$.

For a precise definition of "sufficiently smooth" check out the paper, but we note that a sufficient condition for satisfying this is to be continuously differentiable (https://en.wikipedia.org/wiki/Differentiable_function#Differentiability_classes). On top of the true model we of course also have our model estimate $\hat{y}_n\colon \mathbb{R}^d \to \mathbb{R}$, which has been trained on a training sample of size n. We assume a couple of things about this model:

1. $\hat{y}_n$ is deterministic;
2. $\hat{y}_n$ is continuous;
3. $\hat{y}_n$ converges pointwise to some $\hat{y}\colon \mathbb{R}^d \to \mathbb{R}$ as $n \to \infty$;
4. $\mathbb{E}[\hat{y}_n(x) - \psi(x)]^2 \to 0$ as $n \to \infty$ for every $x \in \mathbb{R}^d$.

Most notable is assumption (4), stating that our model estimate $\hat{y}_n$ will estimate the true model ψ *perfectly* as we gather more data. In other words, we're essentially assuming that we can get *zero training error*. This is fine for most unregularised models (not all though, with linear regression being an example), but as soon as we start regularising then this won't hold anymore. We can avoid assuming (4) if we instead merely assume that

$$(\dagger) \qquad \eta(x) := \lim_{n\to\infty} \mathbb{E}[(\hat{y}_n(x) - \psi(x))^2]$$

exists for every $x \in \mathbb{R}^d$, which would correspond to the **bias** of the model. Here (4) would postulate that the model has no bias at all.

Two types of noise

To estimate the width of our prediction intervals we need to quantify the error sources that are present. Given a new observation $x_0 \in \mathbb{R}^d$ we can write

$$y_0 := y(x_0) = \psi(x_0) + \varepsilon(x_0) = \hat{y}_n(x_0) + \eta(x_0) + \eta_n(x_0) + \varepsilon(x_0),$$

where we define $\eta_n\colon \mathbb{R}^d \to \mathbb{R}$ as $\eta_n(x) := \psi(x) - \hat{y}_n(x) - \eta(x_0)$. This neatly splits the noise around our prediction $\hat{y}_n(x_0)$ into the **model bias** $\eta(x_0)$, **model variance noise** $\eta_n(x_0)$ and the **sample noise** $\varepsilon(x_0)$. We therefore need to estimate the uncertainty of all these types of noise when we're computing our prediction intervals.

Noise #1: Model variance

Let's start by seeing how the authors estimate the model error. Here we're bootstrapping our sample $B \gg 0$ many times, fitting our model on each of them and then generating bootstrapped predictions $\bar{y}_{b,n}(x_0)$ for every $b < B$. We will estimate the mean $\mu(x_0)$ of the distribution of $\hat{y}(x_0)$ by the bootstrap estimate

$$\hat{\mu}_n(x_0) := \frac{1}{B}\sum_{b=1}^{B} \bar{y}_{b,n}(x_0).$$

We can thus center the bootstrapped predictions as $m_b := \hat{\mu}_n(x_0) - \bar{y}_{b,n}(x_0)$. Now note that since we're assuming (†) we get that

$$\mathbb{E}[m_b] = \mathbb{E}[\hat{\mu}_n(x_0)] - \mathbb{E}[\bar{y}_{b,n}(x_0)] \to_{b\to\infty} \mathbb{E}[\psi(x_0) - \eta(x_0)] - \mathbb{E}[\hat{y}_n(x_0)] = \mathbb{E}[\eta_n(x_0)],$$

giving us our estimate of the model variance noise.

Noise #2: Sampling and bias

Next up, we want to estimate the bias $\eta(x_0)$ and the sample noise $\varepsilon(x_0)$. With $\bar{y}_{b,n}$ being the bootstrapped models as above, we define the bootstrap validation residuals

$$\text{val\_error}_{b,i} := y(x_i) - \bar{y}_{b,n}(x_i)$$

for every $b < B$ and every $i < n$ which is **not** in the b'th bootstrap sample. This will then estimate the validation residual $y(x_0) - \hat{y}(x_0)$. We also calculate the training residuals $\text{train\_error}_i := y(x_i) - \hat{y}(x_i)$ for $i < n$. Note that

$$\mathbb{E}_b[\text{val\_error}_{b,i}] \approx \eta(x_i) + \eta_n(x_i) + \varepsilon(x_i) \to_{n\to\infty} \eta(x_i) + \varepsilon(x_i),$$

giving us an estimate of the sum of the sample noise and the bias. This would work equally well asymptotically if we replaced the validation errors with the training errors, so we have to decide which one to choose. It turns out that the training errors will usually be too small as we tend to overfit, so we have to rely on the validation errors somewhat. The validation errors will tend to be slightly too large however, as a bootstrap sample only contains roughly 2/3 of the training data on average, meaning that the predictions will be artificially worsened.

This issue is also pointed out in Section 7.11 in the "machine learning bible", Elements of Statistical Learning (https://web.stanford.edu/~hastie/ElemStatLearn/), and as a comprimise betweeen the training- and validation errors they propose the following " $.632+$ bootstrap estimate", which I'll quickly introduce here. We start by defining the **no-information error rate** as

$$\hat\gamma := \frac{1}{n^2}\sum_{i=1}^n\sum_{j=1}^n (y(x_i) - \hat y(x_j))^2,$$

which is the loss if the inputs and outputs were completely independent. From this we define the **relative overfitting rate** as

$$\hat R := \frac{\text{val\_error} - \text{train\_error}}{\hat\gamma - \text{train\_error}},$$

which is equal to 0 if no overfitting is taking place and 1 if the overfitting equals the no-information value $\hat\gamma - \text{train\_error}$. We then define the weight $\hat w := \frac{.632}{1-.368\hat R}$, varying from $.632$ in case of no overfitting (in which case this estimate is equal to the standard $.632$ estimate) to 1 if there is severe overfitting. Our $.632+$ bootstrap estimate of the distribution of $\varepsilon(x_0) + \eta(x_0)$ is then

$$o_i := (1 - \hat w) \times \text{train\_error} + \hat w \times \text{val\_error}.$$

In practice, computing $\hat{\gamma}$ can be quite computationally expensive if n is large, so instead I chose to estimate this by only considering a random permutation of the $y(x_i)$'s and the $\hat{y}(x_j)$'s.

Prediction interval implementation

The algorithm producing the intervals are now quite simple given the above reasoning: we simply have to compute the set

$$C := \{m_b + o_i \mid b < B, i < n\},$$

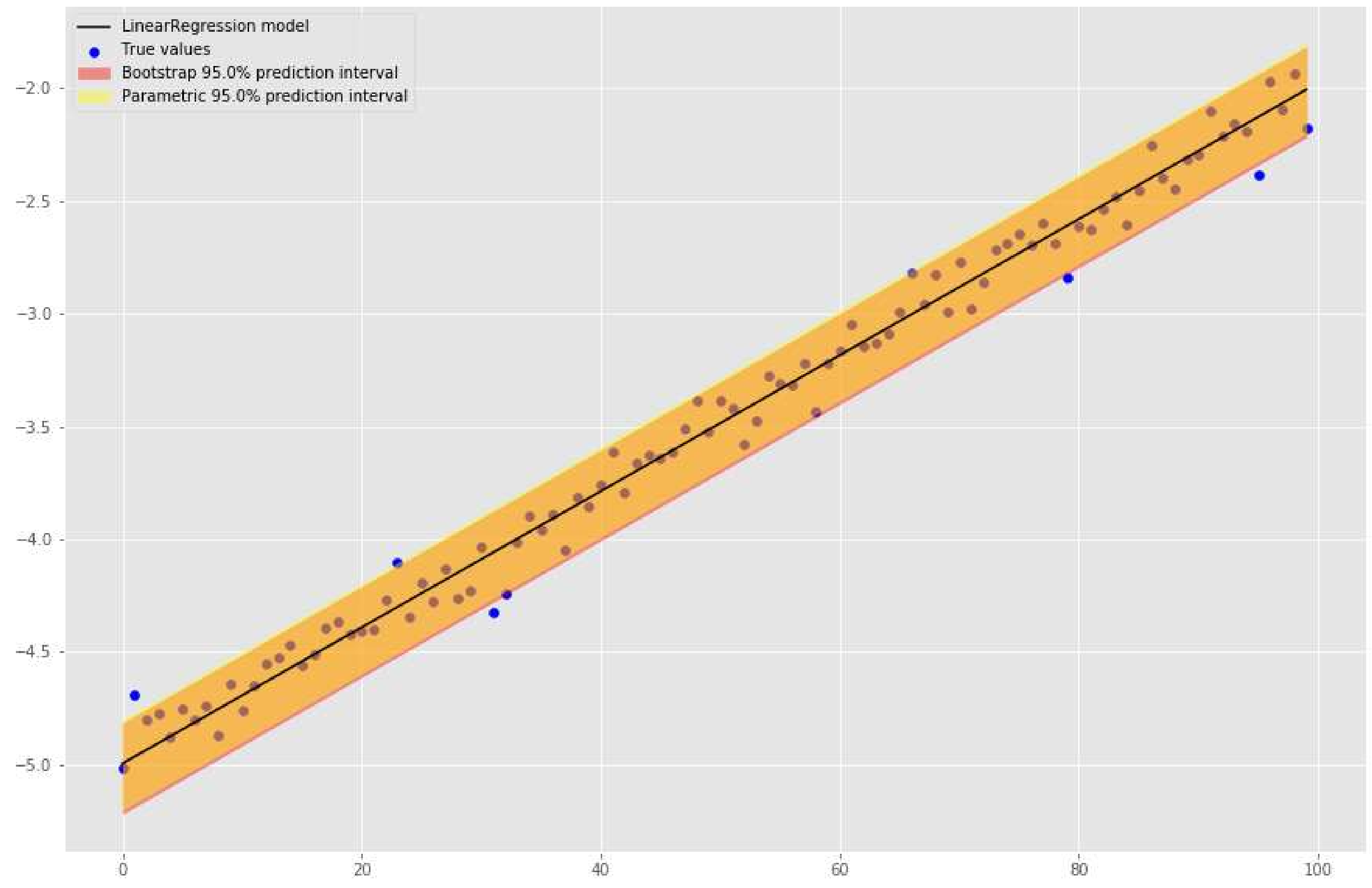
which we showed above is estimating the distribution of $\eta(x_0) + \eta_n(x_0) + \varepsilon(x_0)$, which constitutes all the noise around $\hat{y}_n(x_0)$. From C we can then let our interval be given as the predicted value $\hat{y}_n(x_0)$ offset by the $(100 \cdot \frac{\alpha}{2})$% and $(100 \cdot (1 - \frac{\alpha}{2}))$% percentiles. Here is how we can implement all of this in Python:

```
def prediction_interval(model, X_train, y_train, x0, alpha: float = 0.05):
  ''' Compute a prediction interval around the model's prediction of x0.

  INPUT
    model
      A predictive model with `fit` and `predict` methods
    X_train: numpy array of shape (n_samples, n_features)
      A numpy array containing the training input data
    y_train: numpy array of shape (n_samples,)
      A numpy array containing the training target data
    x0
      A new data point, of shape (n_features,)
    alpha: float = 0.05
      The prediction uncertainty

  OUTPUT
    A triple (`lower`, `pred`, `upper`) with `pred` being the prediction
    of the model and `lower` and `upper` constituting the lower- and upper
    bounds for the prediction interval around `pred`, respectively. '''

  # Number of training samples
  n = X_train.shape[0]

  # The authors choose the number of bootstrap samples as the square root
  # of the number of samples
  nbootstraps = np.sqrt(n).astype(int)

  # Compute the m_i's and the validation residuals
  bootstrap_preds, val_residuals = np.empty(nbootstraps), []
  for b in range(nbootstraps):
    train_idxs = np.random.choice(range(n), size = n, replace = True)
    val_idxs = np.array([idx for idx in range(n) if idx not in train_idxs])
    model.fit(X_train[train_idxs, :], y_train[train_idxs])
    preds = model.predict(x_train[val_idxs])
    val_residuals.append(y_train[val_idxs] - preds)
    bootstrap_preds[b] = model.predict(x0)
  bootstrap_preds -= np.mean(bootstrap_preds)
  val_residuals = np.concatenate(val_residuals)

  # Compute the prediction and the training residuals
  model.fit(X_train, y_train)
```

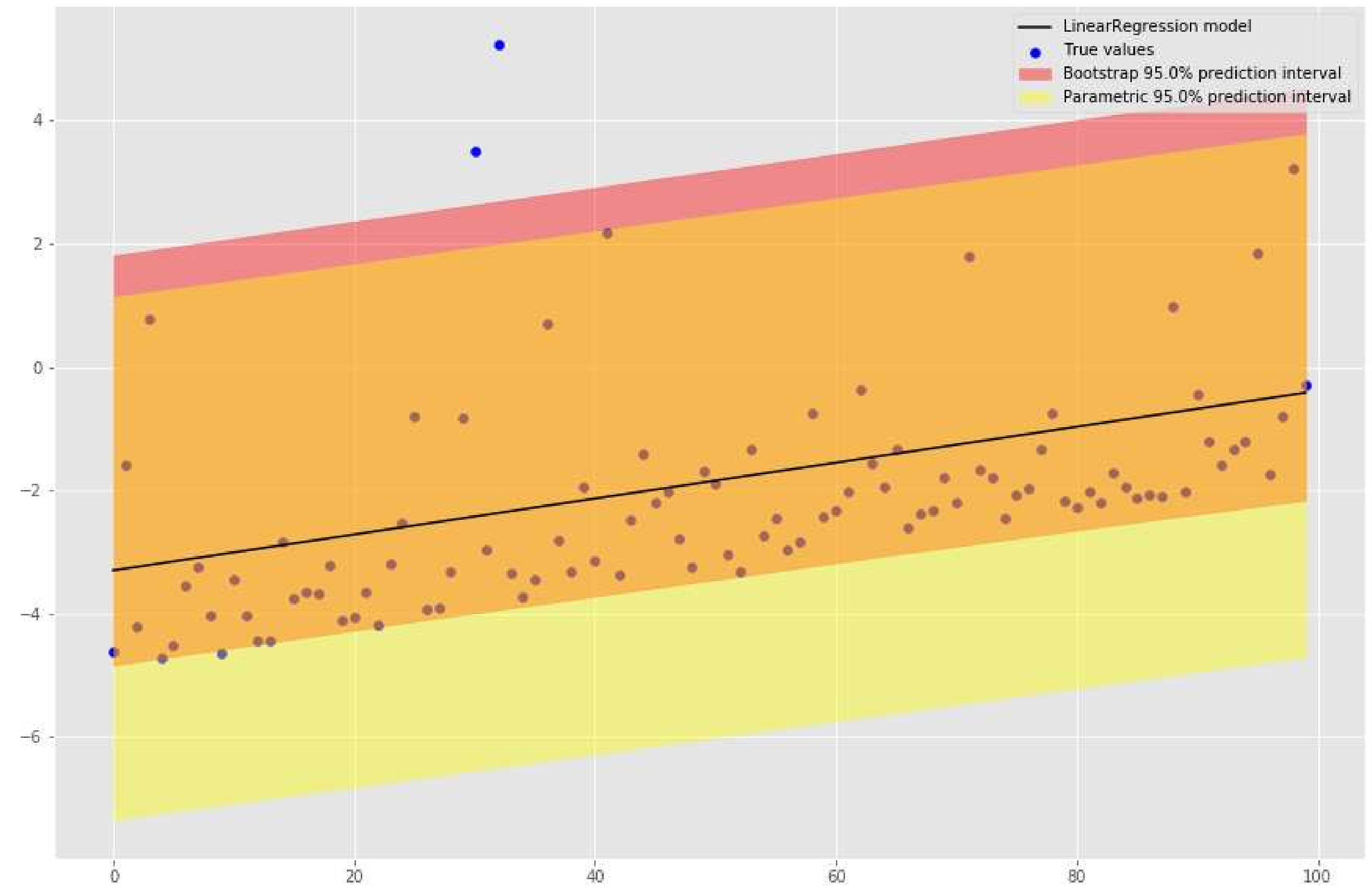
```
    preds = model.predict(X_train)
    train_residuals = y_train - preds

    # Take percentiles of the training- and validation residuals to enable
    # comparisons between them
    val_residuals = np.percentile(val_residuals, q = np.arange(100))
    train_residuals = np.percentile(train_residuals, q = np.arange(100))

    # Compute the .632+ bootstrap estimate for the sample noise and bias
    no_information_error = np.mean(np.abs(np.random.permutation(y_train) - \
      np.random.permutation(preds)))
    generalisation = np.abs(val_residuals.mean() - train_residuals.mean())
    no_information_val = np.abs(no_information_error - train_residuals)
    relative_overfitting_rate = np.mean(generalisation / no_information_val)
    weight = .632 / (1 - .368 * relative_overfitting_rate)
    residuals = (1 - weight) * train_residuals + weight * val_residuals

    # Construct the C set and get the percentiles
    C = np.array([m + o for m in bootstrap_preds for o in residuals])
    qs = [100 * alpha / 2, 100 * (1 - alpha / 2)]
    percentiles = np.percentile(C, q = qs)

    return percentiles[0], model.predict(x0), percentiles[1]
```

Simulations

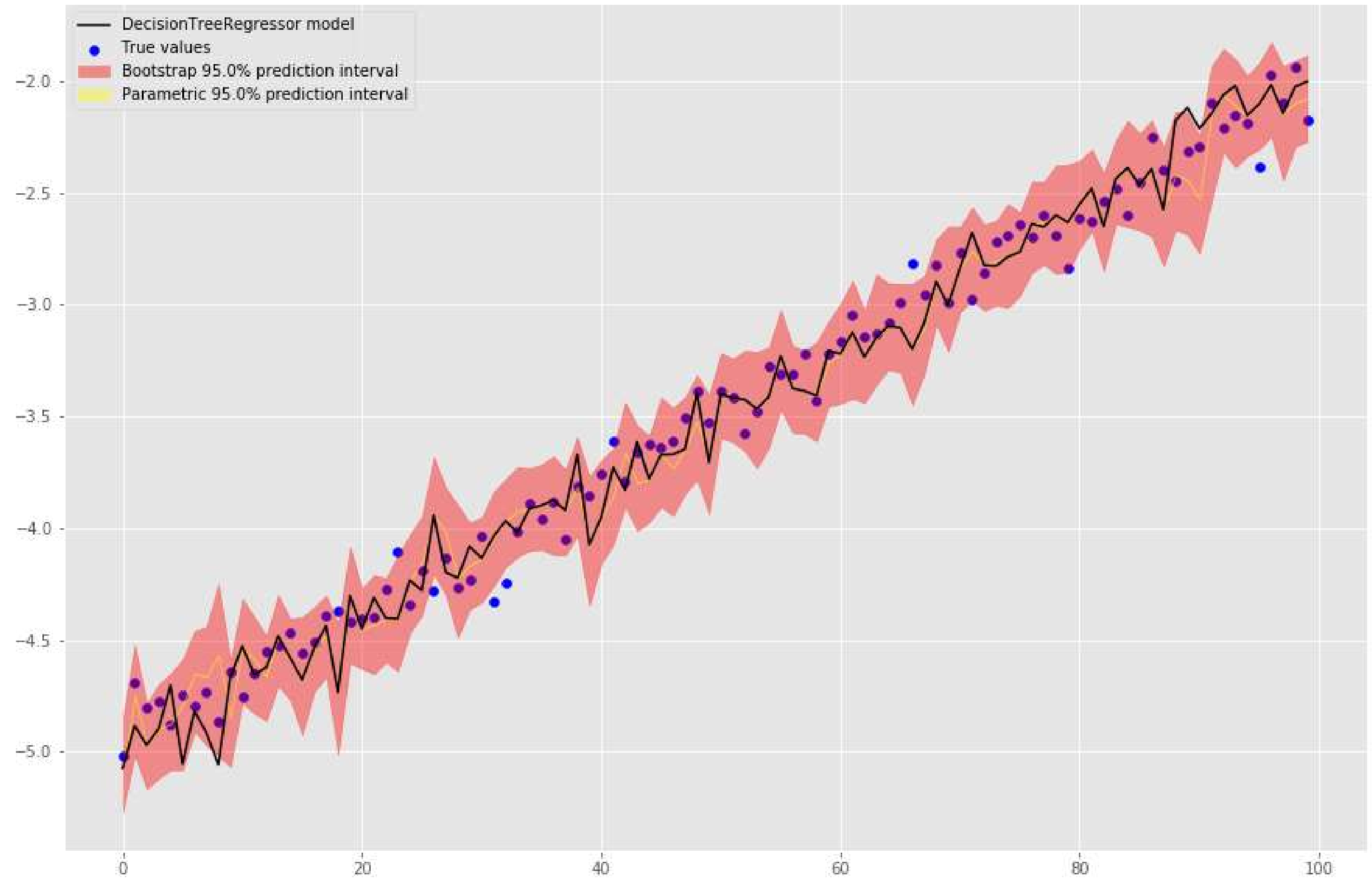
Let’s see how well the above implementation works in practice. Let’s start easy with a linear model, $y(x) := 3x - 5 + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, 0.1)$. Here are two 95% prediction intervals, one computed via the bootstrapping approach and one with the normal theory approach which I covered in the last post (https://saattrupdan.github.io/2020-02-26-parametric-prediction/). Note that we’re showing the *new* values, but instead of working with just a single new value x_0 as above, we’re repeating the above process for all the new values. Here we’re training on $n = 1000$ samples and testing on 100 samples.



In this case the bootstrap interval has a coverage of 95% and the normal theory one having 94%. If we repeat the experiment we see that they are both fluctuating around 95%, sometimes where the bootstrap interval is more accurate and sometimes the normal theory interval being more accurate. Note that in the bootstrapping case we're *not* assuming normal distributed noise, so if we now let $\varepsilon \sim e^Z$ with $Z \sim \mathcal{N}(0, 1)$, i.e. we're assuming that it now follows a log-normal distribution (https://en.wikipedia.org/wiki/Log-normal_distribution) with $\mu = 0$ and $\sigma = 1$, then the bootstrap intervals take the asymmetry into account.



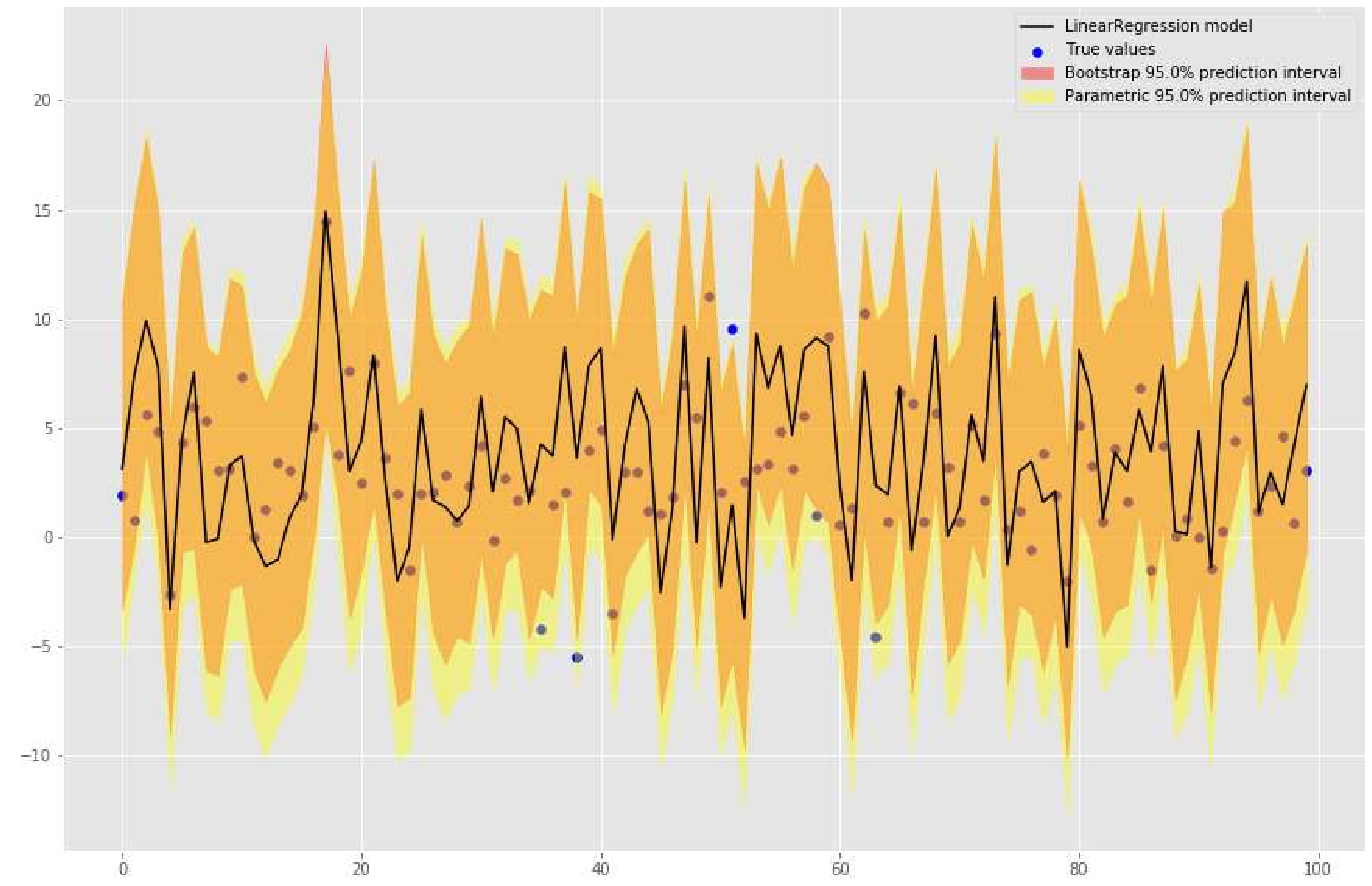
Here we thus get much smaller intervals, and the coverages in this case are 98% and 96% for the parametric- and the bootstrap interval, respectively. Furthermore, if we go to the extreme overfitting case where we instead of linear regression fit a single decision tree, we get the following.



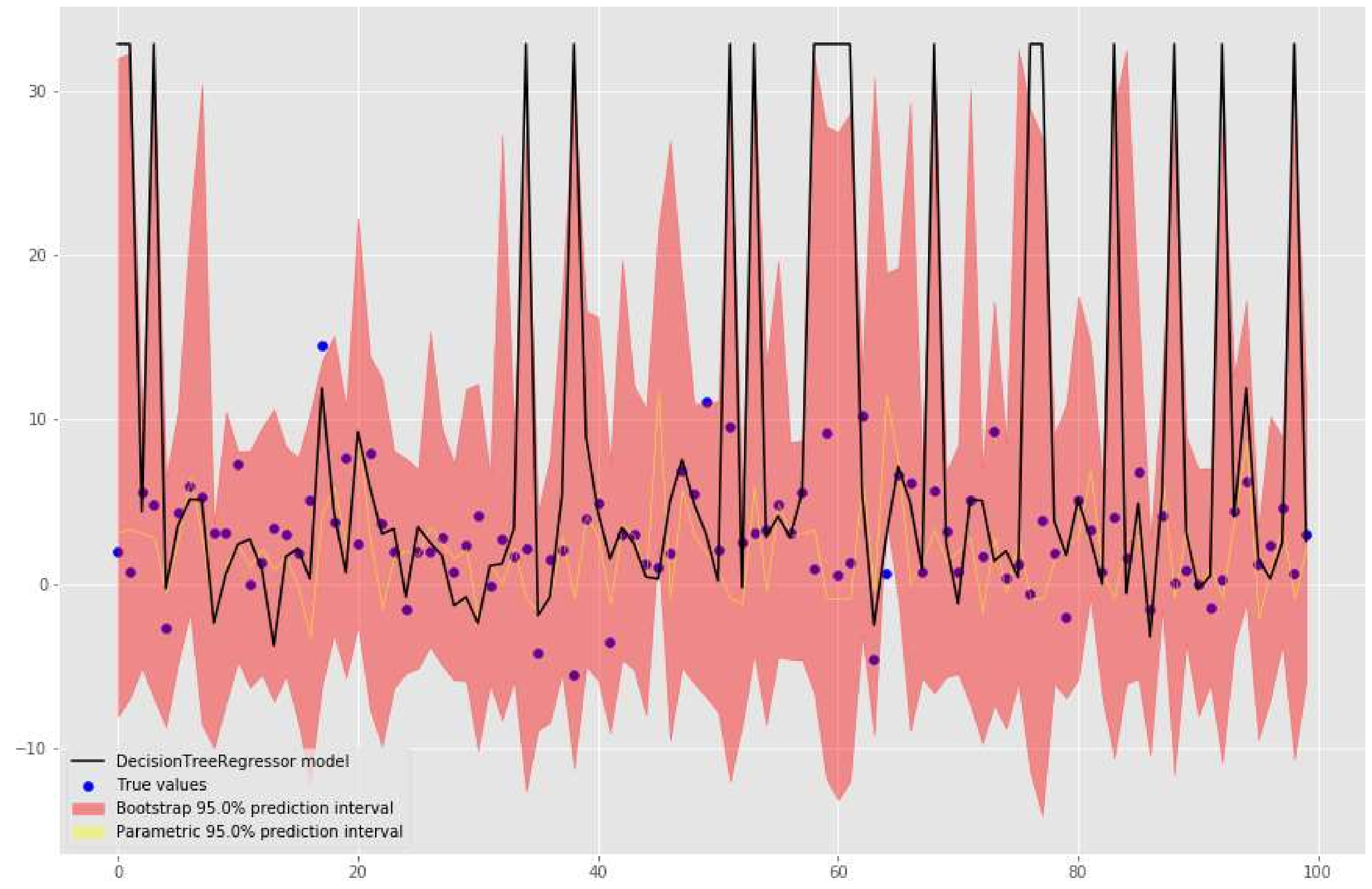
Here the bootstrap interval has a coverage of 92% and the parametric one having a coverage of 1%. Overall, we see that we've really gained something here! We can also test it for non-linear data. Here we've set $d = 5$, i.e. chosen 5 features, and set

$$y(\vec{x}) := e^{x_0} + x_1x_2^2 + \log(|x_3 + x_4|)) + \varepsilon,$$

where ε is multivariate normal with means $\mu_i \sim \mathrm{Unif}(-1, 1)$ and covariances $\mathrm{cov}_{i,j} \sim \mathrm{Unif}(-1, 1)$.



Here the coverage of the normal theory interval is 99% and the coverage for the bootstrap interval is 94%. If we replace the model with a decision tree as before we get the following.



Again we see that the parametric interval has zero width and a coverage of 0%, and the bootstrap interval having a coverage of 96%.

Conclusion

We've produced bootstrapped prediction intervals for almost any predictive model, which is a slight variant of the intervals produced in Kumar and Srivastava (2012) (https://ntrs.nasa.gov/search.jsp?R=20130014367). We've seen that they perform as well as the parametric prediction intervals produced with normal theory on linear data with normal noise, but also that the bootstrapped intervals outperform the parametric intervals when we have non-normal noise, non-linear data or if the model is overfitting.

(htt.

← **PREVIOUS POST (/2020-02-26-PARAMETRIC-PREDICTION/)**

NEXT POST → (/2020-03-09-QUANTILE-REGRESSION/)