3x2 Covariance emulator

September 14, 2017

1 Emulator in the big picture

Our covariance emulator will be quite simple: it will only take in a cosmology and then spit out a matrix to the user. The function signature for making predictions will look something like

```
def predict_matrix(cosmology):
    """
    Inputs:
    cosmology - a container of the cosmological parameters

    Outputs:
    C_inverse - an NxN array containing the inverse covariance matrix.
    The dimensions will be specified by the training data,
    which we have to assume will never change, unless we
    want to get real fancy.
    """
    #do things
    return C_inverse.
```

Internally, the algorithm used to make a prediction with the emulator will look like the following:

1. Predict some small number of important quantities, X, that depend on cosmology Ω such as PCA components of some matrix.
2. Use X to reconstruct the diagonal components D and the off-diagonal components T of the GCD components.
3. Reassemble the GCD components into the inverse covariance matrix C^{-1}.

This algorithm is thus the reverse of section 3 from Morrison & Schneider (2013), where they describe how the emulator is designed. The prediction could be made with Gaussian processes, but doesn't have to be. If, for instance, we observed that some PCA components depended very simply on a certain cosmological parameter then we could model that directly.

Thus, to train our emulator we would apply this prediction code in reverse: take all the input inverse covariance matrices, reduce them to only a few numbers, find a good interpolation of those numbers as a function of cosmology. The following sections will describe how the emulator should be trained.

2 Matrix decomposition: GCD

Following Morrison & Schneider, we should probably use GCD, but this isn't a necessity. The inverse covariance matrix, which is $N \times N$ gets decomposed like (I'm using slightly different notation than in their paper, for convenience)

$$C^{-1} = L^T D L \tag{1}$$

where L is a lower triangular matrix (and therefore has $N(N-1)/2$ indepenent elements) and D is a diagonal semi-positive definite matrix (with N independent elements). Emulating $N(N-1)/2+N$ independent elements isn't feasible, so we have to reduce this further with PCA.

Before moving on, I should note that elements of L can take on any numerical value, but since D is semi-positive then we will work with $\ln D = d$ instead. In other words, we will work with the logarithm of the elements of D. This is important because the Gaussian process wouldn't have knowledge that D is semi-positive, and could yield unphysical predictions.

We also subtract of the mean of all d values and rescale the values of d so that they have variance 1. This looks like

$$d \to \frac{d-\bar{d}}{/}\sigma_d. \tag{2}$$

This makes PCA described next slightly cleaner. When making the prediction, once we have d from the Gaussian processes, we can transform it back to its original form by multiplying on σ_d and adding $\bar{d}$. The same treatment goes for the avlues of L.

3 PCA on the GCD parts

Let's pretend that we have run M different simulations (so for us, $M \sim 16$). Let's think just about d first. If we take all the ds obtained for each M simulation and "stack" them on top of each other into a new matrix $\mathcal{D}$ which is now $N \times M$ (row/column order doesn't matter). Performing PCA on this gives us

$$\mathcal{D} = UBV^T \tag{3}$$

where if $p = \min(N, M)$ then U is a $N \times p$ matrix with $U^T U = I_p$, V is a $M \times p$ with $V^T V = I_p$ and $VV^T = I_M$, and B is $p \times p$ diagonal matrix of singular values. Basis vectors are formed by $\Phi = \frac{1}{\sqrt{M}}UB$ which is $N \times p$, with weights $w = \sqrt{M}V^T$ normalized by $\frac{1}{\sqrt{M}}w^T w = I_M$. The columns of Φ, which are of length N, represent descreasingly important principle components of the original matrix $\mathcal{D}$. This means we only need to use the first few of them. In the Coyote papers they found that using the first five was sufficient. This will be something to test.

The ith column of Φ corresponds to the ith row of w, or w_i. This w_i has mean zero, and (according to the papers...) contains all of the cosmologial dependence, and is also a function of the domain.

In the coyote papers, the domain was k, whereas in our paper we have this very strange behaviour where the domain is not monotonic, since the "full" data vector is a combination of a bunch of data vectors. In other words, looking at Figure 5 of Elizabeth's paper, the data vectors are ξ_+, ξ_-, γ_t, and w. The domain is not monotonically changing along the edge, which isn't viable for design of the emulator. We will have to think of something, like using just bin number or something. The downside of doing so would mean that the number of bins would have to remain fixed forever.

4 Gaussian Process

In this design if we keep five columns then we would utilize five interpolators. We can do this because the weights are all independent of each other, so the prediction by one interpolator is independent of any other interpolator. The challenge we have is to figure out the best way to interpolate w_i across cosmologies and domain. Gaussian processes (GPs) were used in the past because they provide an estimate of the expectation value $\langle w_i(\Omega, x)\rangle$ as well as an estimate of the variance $\mathrm{Var}(w_i)$. Note that we could achieve the same thing if we defined a model (a line, a quadratic, something else) for w_i and knew the posterior distributions of the parameters in our model. Using GPs or such a model means that we can propogate the uncertainty in our estimate forward in a Bayesian way. On the other hand this could be computational expensive and end up being not adding much to a cosomological analysis.

I'll fill the rest of this out later, but the important thing that we care about is picking kernels with hyper-parameters, writing a likelihood, and then finding the best hyperparameters.