

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)

$$C^{-1} = L^T D L \quad (1)$$

where L is a lower triangular matrix (and therefore has $N(N-1)/2$ independent 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.

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 d s 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

$$\mathcal{D} = UB V^T \quad (2)$$

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 $V V^T = I_M$, and B is $p \times p$ diagonal matrix of singular values. Basis vectors are formed by $\Phi = \frac{1}{\sqrt{M}} U B$ 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 decreasingly 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.

4 Gaussian Process