Simulating_correlated_sequence_motifs

May 9, 2016

1 Simulating sequences with correlated and random classes

1.0.1 Motivation

Non-coding genomic regions have been associated with various genomic states, such as DNase hypersensitive sites, transcription factor binding, and histone marks. Each of these genomic states which vary across different cell types is used as the labels to condition the model on common genome sequences. It may or may not be surprising that the labels are highly correlated. But, the correlations in the labels is not assumed in the cost functions of the deep learning models, at least in traditional ones like binary cross-entropy, squared error, etc. Thus, the extent to which the correlated classes influences the efficiency of deep learning has, heretofore, not been explored. In this notebook, I will go over how to generate synthetic datasets of nucleotide sequences with implanted motifs associated with correlated class labels and random class labels. The level of motif information is kept the same between these two datasets for further downstream comparisons.

1.0.2 Simulation overview

Two sets of simulated data will be generated, a so-called correlated labelled dataset and a randomly labelled dataset. The correlation structure of the correlated labelled dataset is taken directly from Basset's DNAse hypsersensitive site dataset, which is a subset of DeepSea's dataset. From this, we also create a randomly shuffled labelled dataset with the same number of labels per sequence to maintain the same level of complexity for a direct comparison. A single, randomly chosen 8nt sequence motif is then associated with each class. A synthetic sequence matrix is then created by implanting the motif's position weight matrix in a non-overlapping fashion for each sequence, depeneding on the class labels. For isntance, if a given sequence has 5 class labels associated with it, then the 5 corresponding motifs are placed on the sequence in random order. The other nucleotide positions given by a uniform distribution, i.e. $p = 1/4$. Then, the cumulative sum of the probabilities is calculated and a uniform random number is drawn for each nucleotide. The corresponding bin that the random number falls in determines the realized nucleotide sequence. Each sequence is only sampled one time.

Importantly, the correlated data and the random data have the same number of motifs, thus they should have the similar levels of complexity, which will allow for a fair comparison.

```
In [1]: import os.path
        import pandas as pd
        import numpy as np
        from six.moves import cPickle
        import h5py
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Load motifs for drosophila melanogaster The implanted motifs comes from Ray et al. "A compendium of RNA-binding motifs for decoding gene regulation" (http://www.nature.com/nature/journal/v499/n7457/abs/nature12311.html). The link to downloaded the motifs is here:

!wget http://hugheslab.ccbr.utoronto.ca/supplementary-data/RNAcompete_eukarya/top10align_motifs.tar.gz

!tar xzf top10align_motifs.tar.gz

Here, each file is a different RBP motif as a position frequency matrix. So, the first step is to compile all of these files into a suitable database. In particular, we can parse each motifs (position frequency matrix) from each file in motifpath (downloaded top10align_motifs folder), create a database (list of arrays), and save as binary format (motif.pickle):

```
In [2]: # load motifs
        motiflist = 'motif.pickle'
        if os.path.isfile(motiflist):

            # load motif list from file
            f = open(motiflist, 'rb')
            motif_set = cPickle.load(f)
            f.close()

        else:
            # download motifs
            motifpath = 'top10align_motifs/'   # directory where motif files are located

            # get all motif files in motifpath directory
            listdir = os.listdir(motifpath)

            # parse motifs
            motif_set = []
            for files in listdir:
                df = pd.read_table(os.path.join(motifpath,files))
                motif_set.append(df.iloc[0::,1::].transpose())

            # save motifs
            f = open(motiflist, 'wb')
            cPickle.dump(motif_set, f, protocol=cPickle.HIGHEST_PROTOCOL)
            f.close()
```

Now, let's load Basset's DNase hypersensitive site data (which is really from ENCODE and Roadmaps Epigenetics project).

```
In [3]: # load DNAse data (courtesy of Basset)
        datapath = '/home/peter/Data/Basset'
        trainmat = h5py.File(os.path.join(datapath,'er.h5'), 'r')
        y_train = np.array(trainmat['train_out'])
        y = y_train.astype(np.float32)
```

To make our toy dataset manageable, let's just take a subset of the Basset's DNase hypersensitive site data, specifically classes 55-105 out of 0-164. This is purely just because smaller datasets are quicker to train, i.e. you only have to wait an hour or so rather than days.

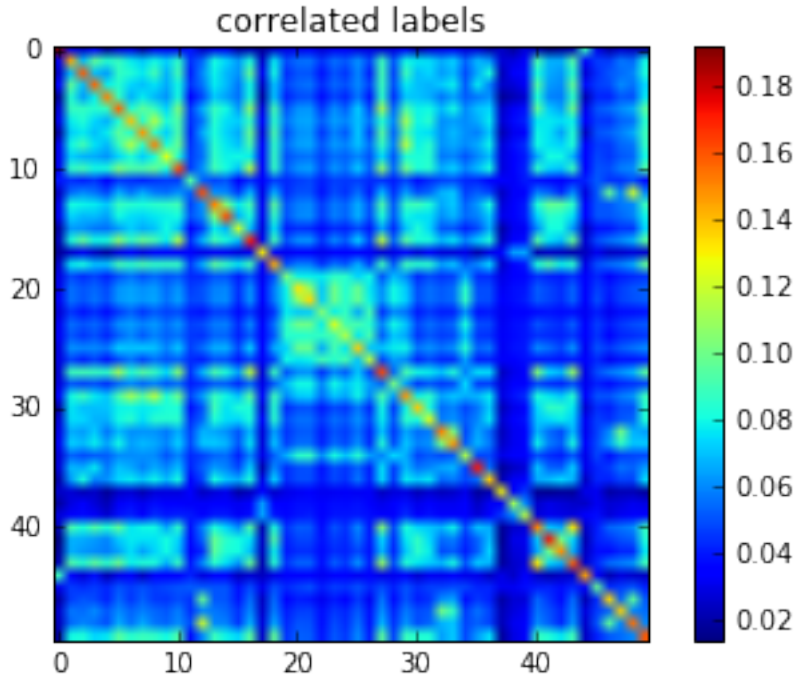
```
In [4]: def data_subset(y, class_range, negative=True):
            "gets a subset of data in the class_range"
            data_index = []
            for i in class_range:
                index = np.where(y[:, i] == 1)
                data_index = np.concatenate((data_index, index[0]), axis=0)
            unique_index = np.unique(data_index)
            num_data = y.shape[0]
            non_index = np.array(list(set(range(num_data)) - set(unique_index)))
```

```
        if negative:
            index = np.concatenate((unique_index, non_index), axis=0)
        else:
            index = unique_index
        return index.astype(int)

    # get a subset of the Basset dataset
    num_seq = 300000        # number of sequences
    class_range = range(55,105)
    index = data_subset(y, class_range, negative=False)
    labels = y[index[0:num_seq],:]
    labels = labels[:,class_range]
```

Let's plot the covariance structure of Basset's labelled data.

```
In [7]: # plot the covariance structure of Basset's dataset
        C = np.cov(labels.T)
        plt.figure()
        plt.imshow(C)
        plt.title('correlated labels')
        plt.colorbar()
```

```
Out[7]: <matplotlib.colorbar.Colorbar at 0x7f7bb29ad0d0>
```



You can see that there is a striking level of covariation between the classes. Moreover, the variance between classes (diagonal elements) are very heterogeneous. This means that this dataset has correlations and heteroskedasticity.

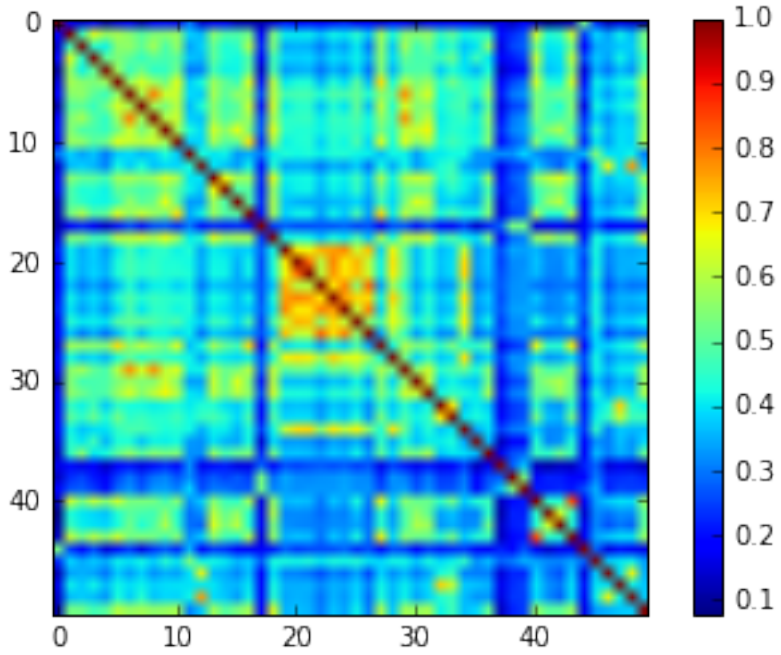
Because the Gauss-Markov conditions are not met, a simple least squares approach is statistically inefficient. Thus, each class prediction should not all be weighed the same way. However, most over-the-counter

cost functions don't account for this, such as binary cross-entropy, which is the cost function of choice for Basset, DeepSea, and DanQ.

Note, this is the covariance of binary variables, so the largest possible covariance is 0.25, i.e. $(x_1 - \bar{x_1})(x_2 - \bar{x_2}) = (1 - 0.5)(1 - 0.5) = 0.25$. We can put it on a normalized scale to show how much corelations there are using correlations instead of covariance:

```
In [8]: C2 = np.corrcoef(labels.T)
        plt.imshow(C2)
        plt.colorbar()
```

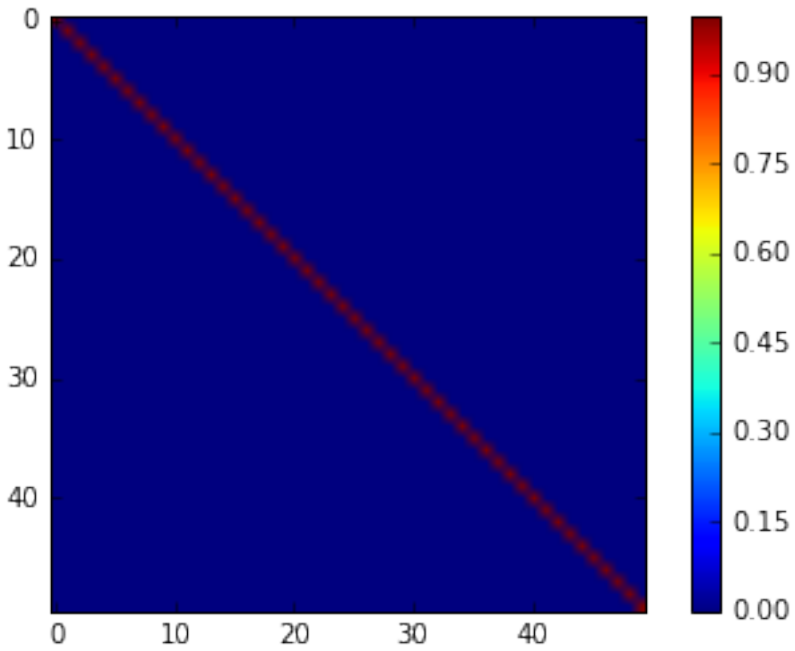
```
Out[8]: <matplotlib.colorbar.Colorbar at 0x7f7bb2804110>
```



One way to combat correlations and heteroskedasticity is by using generalized least squares, which weighs the errors with the inverse covariance matrix: $\arg_\theta \min \left[(y - f(x,\theta))\Gamma^{-1}(y - f(x,\theta)\right]$ This is essentially transforming the correlated data into a new space where it the errors are now generated from an uncorrelated standard normal random variable. To demonstrate this, I will take the Cholesky decomposition of the covariance matrix and apply it to the labels. Then, I will calculate the covariance matrix of the transformed labels.

```
In [9]: L = np.linalg.cholesky(C)
        Linv = np.linalg.inv(L)
        new = np.dot(Linv,labels.T).T
        C = np.cov(new.T)
        plt.figure()
        plt.imshow(C)
        plt.colorbar()
```

```
Out[9]: <matplotlib.colorbar.Colorbar at 0x7f7bb26df390>
```



The covariances are all gone and the variances are all equal to 1. Thus, now the assumptions of least squares holds.

However, for binary data, least squares is an inefficient cost function. And the decorrelation with the Cholesky decomposition transforms the data to a non-binary space. Thus, more efficient cost functions like binary cross-entropy, will not work.

1.0.3 Generate random class labels

To test the performance of deep learning models on datasets with correlated classes and random classes, we need to be fair and generate a comparable level of complexity so as not to create any biases. To do this, I will use the same number of class labels for each sequence, but rather, randomly shuffle the class labels to prevent any correlations from persisting. So, the total number of classes for the random class sequence will be exactly the same as the correlated class sequence, and will thus have a similar level of complexity.

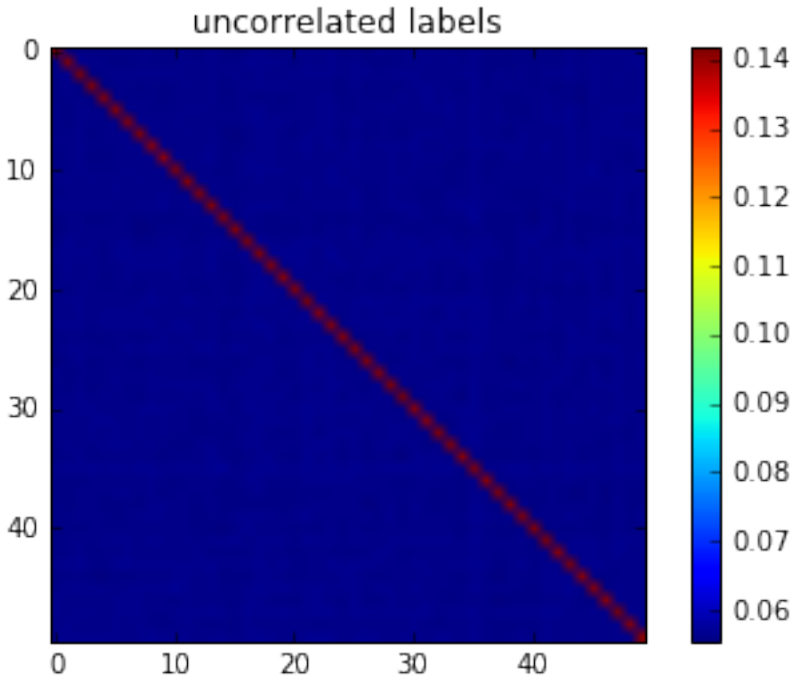
```
In [10]: # determine number of motifs to add to each sequence
         num_motifs = np.sum(labels, axis=1).astype(int)
         num_classes = len(class_range)

         # determine labels
         labels_new = np.zeros((num_seq, num_classes))
         for i in range(len(num_motifs)):
             # randomly permute
             index = np.random.permutation(num_classes)[0:num_motifs[i]]
             labels_new[i, index] = 1
```

Now let's plot the covariance structure of the randomly labelled sequences.

```
In [11]: # plot the covariance structure of the random dataset
         C2 = np.cov(labels_new.T)
         plt.figure()
         plt.imshow(C2)
```

```
plt.title('uncorrelated labels')
plt.colorbar()
```

```
Out[11]: <matplotlib.colorbar.Colorbar at 0x7f7bb25d4ed0>
```



Clearly, there are no correlations and the variances are the same as well. So, this dataset has no correlations and heteroskedasiticity. Now, we need to setup a model for each class. For simplicity, let's just associate each class to a single 8 nucleotide sequence motif, which is randomly sampled from our list of motifs. In this way, if a motif is found, then the class should be predicted. This is certainly overly simplistic, but we may still learn something from doing this simple exercise.

```
In [ ]: # select random subset of motifs from the complete list of motif_set
        motifIndex = np.random.permutation(len(motif_set))[0:num_classes]
        motifs = []
        for index in motifIndex:
            motifs.append(motif_set[index])
```

Let's now generate the sequence and convert it to a one-hot matrix. For a given sequence, the class labels associated with that sequence tell us how many motifs are present. We then shuffle these motifs and implant them on the sequence so that they don't overlap with each other.

```
In [ ]: # loop through each label and generate a sequence
        def geneterate_sequence(label, motifs, seq_length):
            seq = []
            for label in labels_new:

                # find number of classes for given sequence
                # which will correspond to the number of motifs for the sequence
                index = np.where(label==1)[0]
                num_index = len(index)
```

```
            # figure out spacing between motifs
            buffer_size = seq_length - num_index*8
            ave_spacing = np.floor(buffer_size/(num_index+1))

            # generate sequence with motifs
            sequence_pwm = np.ones((4,ave_spacing))/4
            for i in index:
                sequence_pwm = np.hstack((sequence_pwm, motifs[i]))
                sequence_pwm = np.hstack((sequence_pwm, np.ones((4,ave_spacing))/4))
            sequence_pwm = np.hstack((sequence_pwm, np.ones((4,seq_length - sequence_pwm.shape[1]))/

            # convert to one-hot representation
            nucleotide = 'ACGU'
            cum_prob = sequence_pwm.cumsum(axis=0)
            Z = np.random.uniform(0,1,seq_length)
            one_hot_seq = np.zeros((4,seq_length))
            for i in range(seq_length):
                index=[j for j in range(4) if Z[i] < cum_prob[j,i]][0]
                one_hot_seq[index,i] = 1
            seq.append(one_hot_seq)

        # convert to numpy array
        seq = np.array(seq)
        return seq

    seq_length = 600
    seq_corr = geneterate_sequence(labels, motifs, seq_length)
    seq_random = geneterate_sequence(labels_new, motifs, seq_length)
```

Let's split the data into a training set, cross-validation set, and test set with proportions 0.7, 0.15, and 0.15, respectively.

```
In [ ]: def split_dataset(seq, labels, split_size):
            # generate a shuffled subset of data for train, validation, and test

            num_labels = len(np.unique(labels))
            cum_index = np.cumsum(np.multiply([0, split_size[0], split_size[1], split_size[2]],num_seq)
            shuffle = np.random.permutation(num_seq)
            train_index = shuffle[range(cum_index[0], cum_index[1])]
            valid_index = shuffle[range(cum_index[1], cum_index[2])]
            test_index = shuffle[range(cum_index[2], cum_index[3])]

            # create subsets of data based on indices
            train = (seq[train_index], labels[train_index])
            valid = (seq[valid_index], labels[valid_index])
            test = (seq[test_index], labels[test_index])
            return train, valid, test

        # split dataset into training cross validation and testing
        train_size = 0.7
        valid_size = 0.15
        test_size = 0.15
        split_size = [train_size, valid_size, test_size]
```

```
train, valid, test = split_dataset(seq_corr, labels, split_size)
train2, valid2, test2 = split_dataset(seq_random, labels_new, split_size)
```

Now let's save the datasets as an hdf5 file.

In [ ]:
```
def save_dataset(savepath, train, valid, test):
    f = h5py.File(savepath, "w")
    dset = f.create_dataset("trainx", data=train[0])
    dset = f.create_dataset("trainy", data=train[1])
    dset = f.create_dataset("validx", data=valid[0])
    dset = f.create_dataset("validy", data=valid[1])
    dset = f.create_dataset("testx", data=test[0])
    dset = f.create_dataset("testy", data=test[1])
    f.close()

savepath = "synthetic_correlated_motifs_"+ str(num_seq) + ".hdf5"
save_dataset(savepath, train, valid, test)

savepath = "synthetic_random_motifs_"+ str(num_seq) + ".hdf5"
save_dataset(savepath, train2, valid2, test2)
```

In [ ]: