EECE5644
Aleksei Krotov
Homework 3

The code for this and other assignments of this class is in this repository.

Q1.

1. The function "fun(x)" is a $R^5 \to R$ mapping and it processes the first 5 dimensions of an iterable variable *x*, combining them in a heterogeneous second-order polynomial which can be algebraically expressed as
   $f(\mathbf{x}) = \beta^T \mathbf{x} + \mathbf{x}^T B \mathbf{x} + c$, where
   $$\beta = \begin{pmatrix} 2 \\ -1.1 \\ 0.7 \\ 1.2 \\ 0 \end{pmatrix}, \; B = \begin{pmatrix} 0.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.75 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & -0.75 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.7 \end{pmatrix}, \; c = 1.3$$
   This function is not linear, as not all the terms are linear.

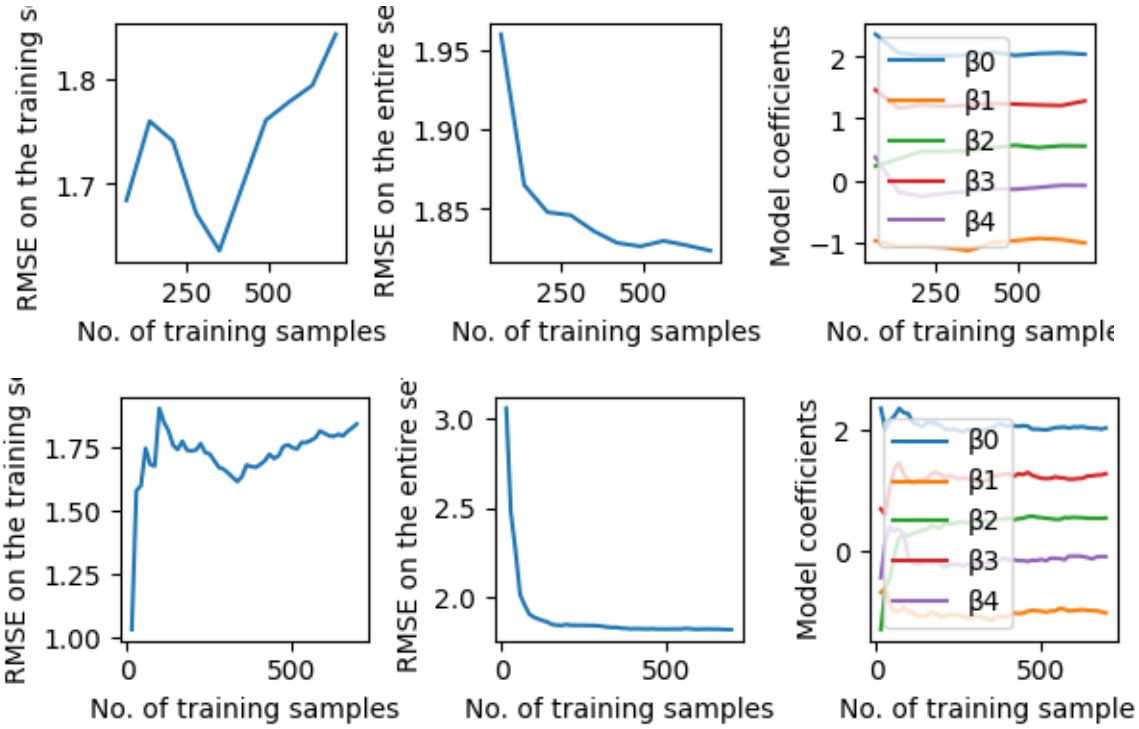
2. The script "data_generator.py" generates pseudo-random five-dimensional real vectors from a normalized Gaussian distribution (zero mean and identity covariance), maps them to real scalars via "fun(x)", and adds random noise from a Gaussian zero-mean distribution with specific standard deviation. The produced random vectors and the corresponding noise-corrupted scalars are saved as python variables with customized naming convention reflecting number of samples, dimensionality, and amount of noise added. All parameters are configurable, including initialization of the random number generator with a given seed for reproducibility, and there are ample log notes for the terminal output.
3. Changing N from 1000 to 2000 would result in generating 2000 random vectors and corresponding noise-corrupted scalars.
4. Changing d from 5 to 10 would result in generating 10-dimensional random vectors, however only the first five components would be mapped onto scalars via function "fun(x)", i.e. components 6-10 would be saved with the variable X but they would be (pseudo-) independent from the variable Y.

Q2.

1. The root mean square error between ground-truth and regression-predicted variables captures the "overall" error of predicting the dataset variables. It is also proportional to the negative log-likelihood of a particular dataset conditioned on regression parameters. Empirically it corresponds to a best-fit line (curve) that human would fit to a set of data points. Conveniently for optimization, that function is convex, for Gaussian zero-mean noise. Therefore, minimizing the RMSE, the sum of squared errors (SSE), or negative log-likelihood $p(D|\beta)$ are all essentially the training objectives of a linear model.
2. In the *sklearn.linear_model.LinerarRegression* class, the method *fit* creates an estimator object using the training data (predictor vectors and output scalars). An estimator object primarily consists of coefficients of a linear regression / linear model. It may be used for predicting the outcomes using the method *predict* for some predictor vectors and its performance can be characterized in various ways. An attribute *coef_* contains coefficients (a vector) whose outer product with predictor vector/matrix produces estimated (predicted)

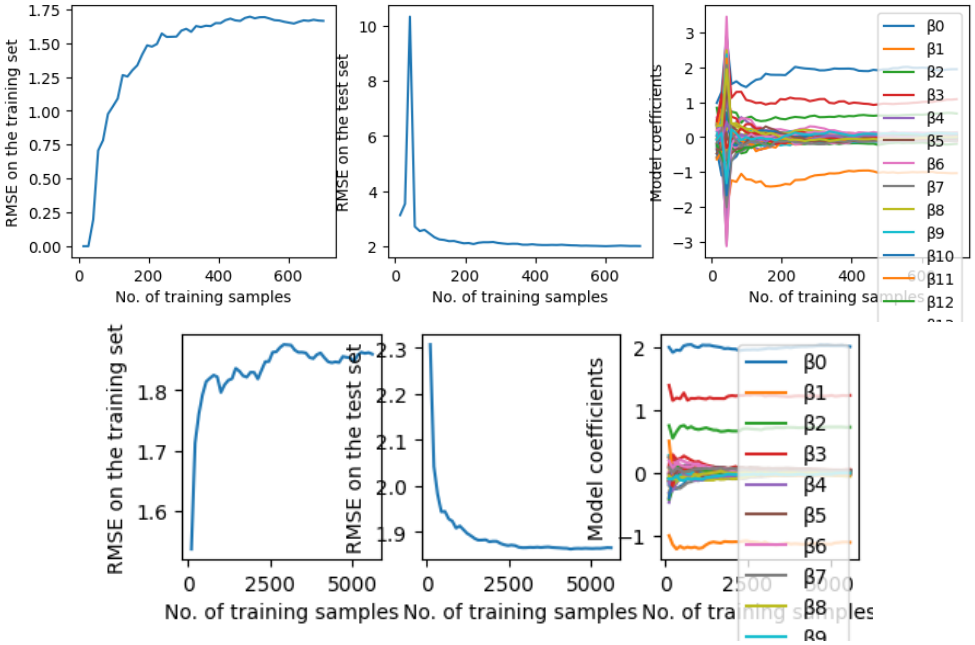
outcome value. The attribute *intercept_* contains a scalar representing a linear shift of the predicted data relative to its center (mean).

3. Using [10, 20, …, 100]% of the training set (700 samples out of 1000 samples in the entire dataset) for fitting the linear model to the generated dataset resulted in varying RMSE in the training subset, RMSE in the training subset, and varying model coefficients (see the Figure below). More specifically, for the next figure, a finer resolution was used to use [2, 4,… 100]% of the training set better highlighting the trends. It reveals that the RMSE in the training dataset first increased with the number of samples used, then seemingly plateaued and the fluctuations were likely caused by pseudo-random nature of the samples. Counter to that, the RMSE in the entire dataset decreased monotonically, but asymptotically, to approximately the value 1.8, with the number of training samples. If the process producing the data was linear, the RMSE would approach the standard deviation of the noise in the data, i.e. $\sigma = 0.1$. In this example, the process producing the data is nonlinear and the linear regression model fails to account for that, therefore increasing the number of samples would not considerably improve the regression. However, the generated predictors are limited to a relatively small domain $N(\boldsymbol{\mu} = 0, \boldsymbol{\Sigma} = I)$ where nonlinearities have a small effect, thus the RMSE decreased monotonically with the number of samples. The coefficients exhibited fluctuations at small number of training samples, then converged to the values, of similar magnitude to the coefficients used for generating the label data.



4. Increasing dimensionality of the predictors from five to 40 while the label data only used the first five components of the predictor variable resulted in different relation of RMSE and number of training samples. That RMSE increased rapidly, then plateaued close to 1.7. The RMSE on the test set showed a spike at about 40 test samples, but then rapidly decreased and plateaued close to 2. Both 1.7 and 2 are likely related to a combination of noise variance in the data and the variance resulting from nonlinearity of the underlying

process. The spike might be related to the number of samples reaching the dimensionality 40 of the predictor when estimation of the coefficients resulted in a numerical artifact. The LM coefficients fluctuated at small sample size, spiked around the same value 40 of test sample size, then the first five coefficients converged to approximately the same values as in the previous example, while the other coefficients, corresponding to nuisance components of the linear regression, fluctuated around zero. Increasing the number of samples would not considerably decrease the RMSE or coefficient estimates, because the same two factors – noise variance and nonlinear relation between the predictor and true labels – would limit the estimator's accuracy. A quick check with 8000 samples at the same dimensionality 40 has confirmed that (second Figure)



Q3.

1. Just as the function *fun* can be expressed as a linear combination of a vector *x* and its quadratic form, the lift of dimensionality with pairwise products can be expressed as vectorizing the upper-triangular matrix of an outer product of vector *x* with itself.

```
import numpy as np
def lift(x):
    d = len(x)
    XOutProd = np.outer(x,x)
    return  np.concatenate((x,XOutProd[np.triu_indices(d)]))
```

2. The function *fun* can be expressed with respect to the vector *x'* as $f(x') = \beta'^{T}\mathbf{x} + c$ by matching the coefficient β' to the coefficient β, the quadratic form $\mathbf{B}$, and the free term b from the function *fun*, manually adjusting the order of the components:

$\beta' = \begin{pmatrix} \beta \\ \beta'' \end{pmatrix}$, where β'' can be obtained as vectorized upper-triangular matrix of $2\mathbf{B} - diag(\mathbf{B})$, as in the following code snippet

```
def fun_prime(x):
    beta = np.zeros((len(x),1))
    beta = [2, -1.1, 0.7, 1.2, 0]
    B = np.zeros((len(x),len(x)))
    B[0,0] = 0.4
    B[4,4] = - 0.7
    B[3,1] = - 0.75
    B[1,3] = - 0.75
    c = 1.3
    # Y = beta @ x.T + x @ B @ x.T + c # Verified numerically.
    beta2 = 2 * B - np.diag(np.diag(B))
    beta2 = beta2[np.triu_indices(B.shape[0])]
    beta1 = np.concatenate((beta,beta2))
    xLift = lift(x)
    return beta1 @ xLift.T + c
```

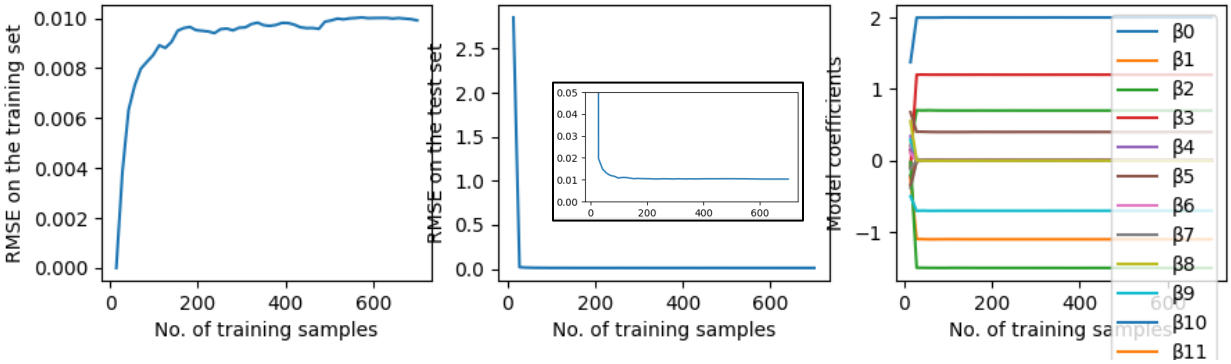
No formal algebraic manner to express the lift of dimensionality was found, hence no proof of equivalence is provided. The equivalence of the function *f'(x')* to *f(x)* was verified with a few arbitrary vectors *x*.

3. Lifting dimensionality of every row of a matrix dataset can be simply implemented by using the function *lift*(x) in a loop.

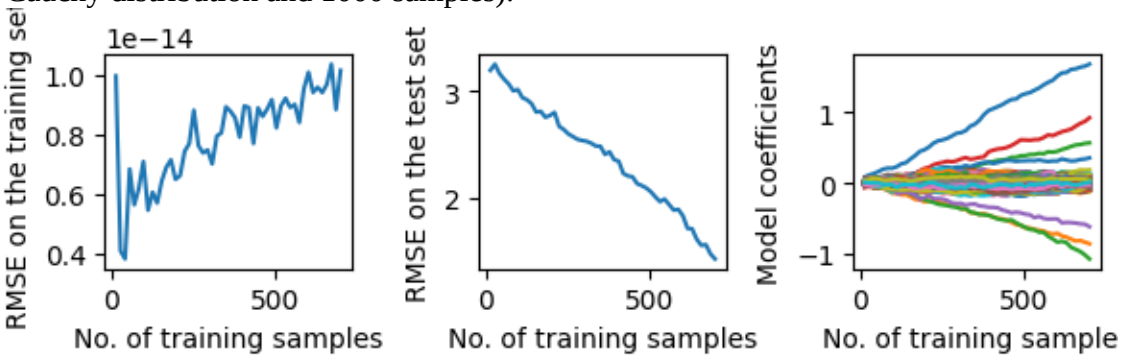
```
def lift_Dataset(x):
    """Lift dimensionality for each row of a matrix dataset x [N x d]"""
    # Assuming correct format of the input
    N = x.shape[0]
    d = x.shape[1]
    d1 = d * (d + 3) / 2 # d linear terms, d quadratic terms, d(d-1)/2
combinations at i != j
    X1 = np.full((N,d1), np.nan)
    for i in range(N):
        X1[i,:] = lift(x[i,:])
    return X1
```

4. Lifting dimensions of the predictors generated by data_generator to include first-order pairwise products resulted in a better-behaved relation of RMSE and coefficients with the number of training samples.
   As this next figure shows, the RMSE on the training set increased with the number of samples and reached saturation close to 0.01, and, counter to that, the RMSE on the test set decreased asymptotically to 0.01. The model now includes the non-linear predictors that used to be neglected in Q2, therefore RMSE approaches the standard deviation of noise.
   Out of 20 coefficients, only seven were non-zero and they matched the corresponding coefficients from β, and B. The intercept 1.30010 was read from the model.intercept_

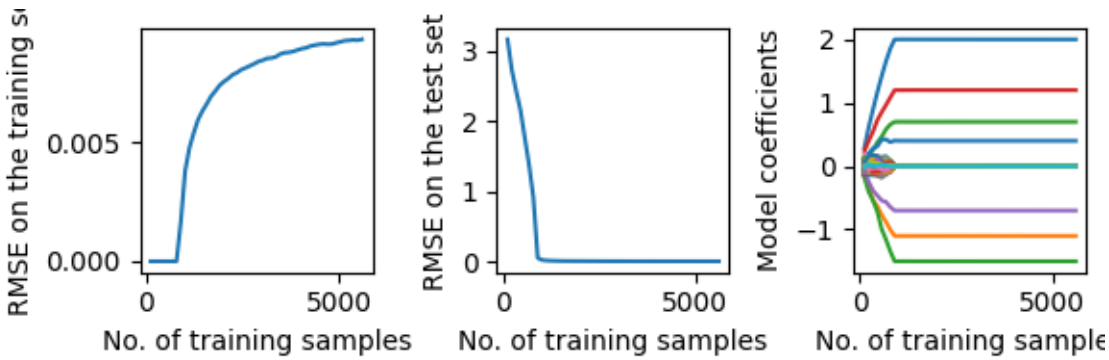
attribute and is also sufficiently similar to the generator's intercept 1.30. Function *fun* has been adequately estimated by this approach.



5. Introducing nuisance predictors via setting generated d to 40, and additionally lifting all the predictors, resulted in a different behavior of the model. While RMSE on the training set was negligibly small and slowly increasing, the RMSE on the test set was relatively large and was slowly decreasing. While some coefficients stayed close to zero, the same subset of potentially "true" coefficients showed linear growth. All the plots appear like the leftmost parts of the previous ones. These dataset and model have not allowed to adequately estimate the function *fun*, due to model overfitting (860 predictors from either Normal or Cauchy distribution and 1000 samples).



Increasing the number of samples to N=8000 resulted in the model improvement and in a behavior similar to the Q3.4. Moreover, the test RMSE reached plateau and the near-zero coefficients approached zero exactly when the training sample size reached 860.



Q4.

1. The function Lasso_CV reads the (X,y) dataset generated by data_generator in the same format as before in this assignment; Instead of using a linear regression model, it uses an L1-regularized linear regression model, a.k.a. Lasso, whose minimization objective is

$$C = \frac{\sum_{i=1}^{N}(y-\hat{y})^2}{2N} + \alpha\,|\beta|_1$$

   The function then uses a 5-fold cross-validation, i.e. splits the training subset (70% of the entire dataset) into 5 subsets, and iteratively uses each one for validating and the remaining four – for training the Lasso regression model. The function is finally trained on the whole training subset and tested on the testing subset.

2. The function swept through α values between 2^{-20} and 2^{20}, as the value 2^{-10} did not allow to observe the overfitting effect with minimum regularization. The plots below show that at small α, RMSE was close to 0.8 indicating potential overfitting. Counter to the Linear Regression model, the coefficients were close to the correct values and not to zero. Reaching $\alpha = 2^{-10\pm0.08}$ minimized the RMSE close to 0.01, matching the amount of noise in the data, most of coefficients became zero, and the non-zero coefficients matched closely those in the generator function: intercept 1.299 and the coefficients $[1.999, -1.100,\ 0.699, 1.199,\ 0.400, -1.500, -0.699]$. Further increasing alpha resulted in reduction of coefficients and rapid clamping of them at zero, as RMSE reached and plateaued at 3. Using Lasso regularization under optimized regularization factor allowed learning the adequate model even for a limited sample size, unlike in Q3.5 for N=1000.