MLPR Assignment 2 - Predicting CT Slice Locations

s1866666

November 2018

1 Getting Started

Python will be used for code demonstration throughout this assignment report. The following is a simple set-up for this work, in which necessary libraries and data set are loaded.

```
from ct_support_code import *
from scipy.io import loadmat
import matplotlib.pyplot as plt
data = loadmat("ct_data.mat", squeeze_me=True)
X_test_raw = data["X_test"]
X_train_raw = data["X_train"]
X_val_raw = data["X_val"]
y_test = data["y_test"]
y_train = data["y_train"]
y_val = data["y_val"]
```

1.a Mean of positions

Computing the mean of the training positions in `y_train` by `np.mean(y_train)` outputs `-9.14e-15`. This verifies the mean of the training positions is zero ignoring rounding errors.

The following codes computes the mean with standard error of the 5,785 validation positions and the first 5,785 out of 40,754 training positions.

```
N = y_val.shape[0]
mean_y_val = np.mean(y_val)
mean_std_y_val = np.std(y_val)/np.sqrt(N)
mean_y_train = np.mean(y_train[:N])
mean_std_y_train = np.std(y_train[:N])/np.sqrt(N)
```

Printing the results gives that the mean of validation positions is -0.216 ± 0.013, and the mean of the first 5,785 training positions -0.442 ± 0.012. This shows the mean of validation positions differs from the mean of the entire training positions (which is 0) by roughly 0.2. It also shows that the mean of the entire training positions lies far outside 2 standard deviation of the mean of the first 5,783 positions of it. These imply that the positions are not identically distributed, that is, samples are not drawn from one distribution. Therefore the standard error bars do not reliably indicate the average of locations in future CT slice data. After all, It's not very reasonable to think that every patient should have identical locations scanned.

1.b Input modification

The following code finds the constant and replicate features in the training inputs and discard them from training, validation and test inputs.

```
# find and discard const columns in X_train
is_const = np.all(X_train_raw==X_train_raw[0,:],0)
const_idx = np.where(is_const)[0]
X_train = X_train_raw[:,np.logical_not(is_const)]

# find and discard duplicate columns in X_train
uni, unique_idx = np.unique(X_train, axis=1,return_index=True)
repeat_idx = np.setdiff1d(np.arange(X_train.shape[1]),unique_idx)
X_train = X_train[:,np.sort(unique_idx)]

# discard const columns in X_val and X_test
const_mask = np.ones(X_val_raw.shape[1],dtype=bool)
const_mask[const_idx] = False
X_val = X_val_raw[:,const_mask]
X_test = X_test_raw[:,const_mask]

# discard duplicate columns in X_val and X_test
repeat_mask = np.ones(X_val.shape[1],dtype=bool)
repeat_mask[repeat_idx] = False
X_val = X_val[:,repeat_mask]
X_test = X_test[:,repeat_mask]
```

The constant column indices (0-based) are: ¡59, 69, 179, 189, 351¿, and duplicate column indices after removing the constant columns are: ¡76, 77, 185, 195, 283, 354¿. There are 373 features remained after this modification.

2 Linear regression baseline

To fit a linear regression model with the training data, the following function is defined.

```
def fit_linreg(X, yy, alpha):
    """ Fits a linear regression XW+b=y with L2 regularization"""
    X_bias = np.concatenate([np.ones((X.shape[0],1)), X], axis=1)
    reg = np.eye(X_bias.shape[1])*np.sqrt(alpha)
    reg[0,0]=0
    X_reg = np.concatenate([X_bias, reg], axis=0)
    yy_reg = np.concatenate([yy, np.zeros(X_bias.shape[1])])
    ww_bias = np.linalg.lstsq(X_reg, yy_reg, rcond=None)[0]
    return ww_bias[1:], ww_bias[0]
```

And we then fit the model and evaluate the root mean square errors of training and validation set as follows:

```
def rtmsq_error(ww, bb, X, yy):
    """ Computes the root mean square error of y and the prediction XW+b"""
    res = yy-(X.dot(ww)+bb)
    return np.sqrt(res.dot(res)/X.shape[0])

alpha = 10
ww_lstsq, bb_lstsq = fit_linreg(X_train, y_train, alpha)
e_train_lstsq = rtmsq_error(ww_lstsq, bb_lstsq, X_train, y_train)
e_val_lstsq = rtmsq_error(ww_lstsq, bb_lstsq, X_val, y_val)
```

function	E_{train}	E_{val}
`fit_linreg`	0.3557589	0.4205925
`fit_linreg_gradop`	0.3557597	0.4206038

Table 1: Errors obtained using different optimizer

Using the same procedure we fit another model using '`fit_linreg_gradop`' in the support code and evaluate training and validation error. Shown in table 1 are the errors obtained by the two model. We can see that the training and validation errors of the model fitted by `fit_linreg` are slightly lower than the one fitted by `fit_linreg_gradop`. This is because `np.linalg.lstsq` in `fit_linreg` directly solves for the closed form solution of least square, whereas `fit_linreg_gradop` finds the solution using gradient descent, which always finds a solution near the global minimum though, can hardly get to the exact point of the minimum given the limited number of iterations to perform. The gradient with respect to the parameters approaches to 0 when the parameters approach the minimum, thus it's most likely either that the parameters converge to the minimum too slowly, or overshoot the minimum, depending on the learning rate.

3 Decreasing and increasing the input dimensionality

3.a Principal Components Analysis

Suspecting the presence of overfitting issue of the baseline model in section 2, which includes all 373 features in linear regression, we will now try to reduce the input dimensionality to K by using Principal Components Analysis (PCA). The following code reduces the number of features in the training and validation input to K=10, and fits a linear regression model with the reduced training data.

```
X_train_cen = X_train - np.mean(X_train,0)
X_val_cen = X_val - np.mean(X_train,0)

K = 10
alpha = 10
V = pca_zm_proj(X_train_cen, K=K)
X_train_red = X_train_cen.dot(V).dot(V.T)+np.mean(X_train,0)
X_val_red = X_val_cen.dot(V).dot(V.T)+np.mean(X_train,0)
ww,bb = fit_linreg(X_train_red, y_train, alpha)
```

Using the same procedure, another model with $K = 100$ features is fitted.

The training error of these two models is likely to go up because the reduction of input dimension decreases the fitness to the training data, thus generating more residuals in training error. Specifically, when there are many features, parameters are more flexible in that a certain change can have less effect on the loss. Therefore the baseline model is able to fit the outliers better and thus has a lower training error.

Shown in table 2 is the training and validation errors of these two models with $K = 100$ and $K = 10$, along with the errors of the baseline model in section 2 for comparison.

K	E_{train}	E_{val}
373 (baseline)	0.3558	0.4206
100	0.4106	0.4328
10	0.5739	0.5717

Table 2: Errors of models with different input dimensionalities

The result shows when the dimension of input gets smaller the validation error get higher and the training and validation errors get closer. These indicate the models with reduced input turn out to underfit the data. We can do a exhaustive search for the optimal number of features. Figure 1 shows the training and validation errors of models trained with K ranging from 10 to 370. The search interval is 10. The lowest validation error is 0.4206, achieved at K=370, which is still higher than the baseline

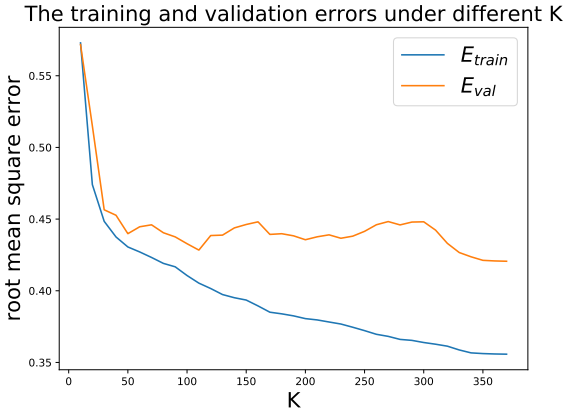


Figure 1: The training and validation errors against dimension of input

model (K=373), meaning that the baseline model does not overfit.

3.b Augmented input

Shown in figure 2 is the histogram of the values of the 46th feature in the training set. the values are divided into 10 bins. We see from the histogram that most of the values are located about 0 and the amount to the left of this majority is roughly the same as the amount to the right. 80.36% of the

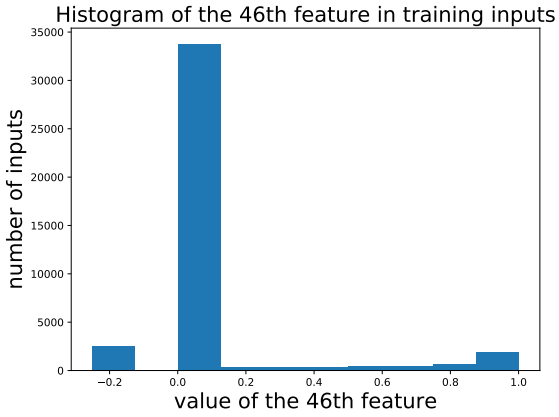


Figure 2: The histogram of the values of the 46th feature in the training set

values in the entire training inputs are equal to -0.25 and 0, computed using the code below.

```
fraction = np.mean((X_train == -0.25)+(X_train == 0.0))
```

We will now try to add more features to the input by adding binary features. In particular, the inputs are divided into three classes by their values, namely $x > 0$, $x = 0$, and $x < 0$, where x is a certain input value. Using one-hot encoding, each feature will produce three more features with the boolean values depending on the class it belongs to. Only two of these new features need to be added for each original feature, because the third one is determined by the two. The code below augments the input dimension using the said procedure and fits a linear regression model, meaning we are now fitting a model with 1119 input features.

```
aug_fn = lambda X: np.concatenate([X, X==0, X<0], axis=1)
alpha = 10
ww, bb = fit_linreg(aug_fn(X_train), y_train, alpha)
e_train = rtmsq_error(ww, bb, aug_fn(X_train), y_train)
e_val = rtmsq_error(ww, bb, aug_fn(X_val), y_val)
```

The training error of this model is likely to go down, because more features come with more flexibility of the parameters, enabling the model to fit more precisely to the underlying properties or merely the noise of the data. Although using too many features poses a risk of overfitting the data, in that special cases are built for outliers, it sometimes improves the model that is overly simple by introducing more complexity to it.

The training and validation errors of this augmented model are shown in table 3, along with the ones of baseline model for comparison. We see that the validation error goes down, meaning that the augmented model fits the data better. This also indicates that our baseline model in section 2 underfits the data.

Model	E_{train}	E_{val}
Baseline	0.3558	0.4206
Augmented	0.3178	0.3770

Table 3: Errors of the augmented model along with the baseline

4 Invented classification tasks

In section 3.a, we reduced the dimension of the input using PCA, we will now try to "pre-classify" the inputs into classes and then perform linear regression to fit the propabilities of these classes. The following code fits a model that first classifies the input into 10 classes, and uses the probability of these 10 classes to predict the output.

```
def fit_logreg_gradopt(X, yy, alpha):
    D = X.shape[1]
    args = (X, yy, alpha)
    init = (np.zeros(D), np.array(0))
    ww, bb = minimize_list(logreg_cost, init, args)
    return ww, bb

K = 10 # number of thresholded classification problems to fit
mx = np.max(y_train); mn = np.min(y_train); hh = (mx-mn)/(K+1)
thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)
W = np.zeros((X_train.shape[1],K))
B = np.zeros(K)
for kk in range(K):
    labels = y_train > thresholds[kk]
    W[:,kk], B[kk] = fit_logreg_gradopt(X_train, labels, alpha)
```

```
X_train_class = 1/(1 + np.exp(-(X_train.dot(W)+B)))
X_val_class = 1/(1 + np.exp(-(X_val.dot(W)+B)))
alpha = 10
ww, bb = fit_linreg(X_train_class, y_train, alpha)
e_train = rtmsq_error(ww, bb, X_train_class, y_train)
e_val = rtmsq_error(ww, bb, X_val_class, y_val)
```

Shown in table 4 are the training and validation errors of this model, along with the PCA model with $K = 10$ in section 3.a for comparison. The result shows this model outperforms the PCA model in our

Model	E_{train}	E_{val}
NN (pre-classifed)	0.1383	0.2521
PCA ($K = 10$)	0.5739	0.5717

Table 4: errors of pre-trained neural network along with PCA model($K = 10$)

case.

The dimensional reduction method in this model is different from the PCA model, in that it encodes all the features into fewer dimensions, reserving all the information the input originally has, whereas PCA discards the less important features, thus losing the information which could be useful in predicting the output. Nevertheless, both methods can be useful in easing the overfitting problem and compressing the data, depending on the application.

5 Small neural network

In section 4, we fitted a neural network seperately, that is, the hidden layer nodes are fitted with pre-defined loss functions before the final layer. Now we will train a neural network jointly, meaning there is only one final loss function, and the hidden layer is trained using back-propagation. The codes below defines the functions to train a neural network with one hidden layer and to evaluate the root mean square error of the model.

```
def fit_nn_gradopt(X, yy, K, alpha, init):
    D = X.shape[1]
    args = (X, yy, alpha)
    return minimize_list(nn_cost, init, args)

def nn_rtmsq_error(params, X, yy):
    pp = nn_cost(params,X)
    res = yy-pp
    return np.sqrt(res.dot(res)/X.shape[0])
```

Using these functions, a neural network can be trained as follows. The weights of the first layer is initialized using the fits in section 4.

```
K=10
D = X_train.shape[1]
alpha = 10
init = (np.zeros(K), np.array(0), W.T, B)
params_fit = fit_nn_gradopt(X_train, y_train, K, alpha, init)
e_train = nn_rtmsq_error(params_fit, X_train, y_train)
e_val = nn_rtmsq_error(params_fit, X_val, y_val)
```

In order to break the symmetry of the parameters, we cannot initialize the weights in the hidden layer to be all zero, otherwise the model won't learn because the parameters always have the same gradient. We can initialize the weights using standard normal distribution $\mathcal{N}(0, 1)$:

```
init = (np.zeros(K), np.array(0), np.random.randn(K,D),np.zeros(K))
```

Shown in table 5 is the training and validation errors of neural networks with different weights initialization, along with the model in section 4 for comparison. Due to the variance from random generation in the model using normal initialization, the mean with 1 std. deviation over 10 trials is reported. The result shows that initializing weights using normal distribution and using the fits in section 4 are

Model	E_{train}	E_{val}
Pre-trained (sec. 4 model)	0.1383	0.2521
sec. 4 init.	0.1025	0.2697
normal init.	0.1241 ± 0.001	0.2692 ± 0.002

Table 5: Errors of neural networks with different weights initialization

similar in terms of validation error. However, initialization with normal distribution fits the training data worse in that the training error is relatively higher.

Moreover, it turns out that fitting the neural network jointly performs worse than the model in section 4 in terms of validation error, although it produces lower training errors.

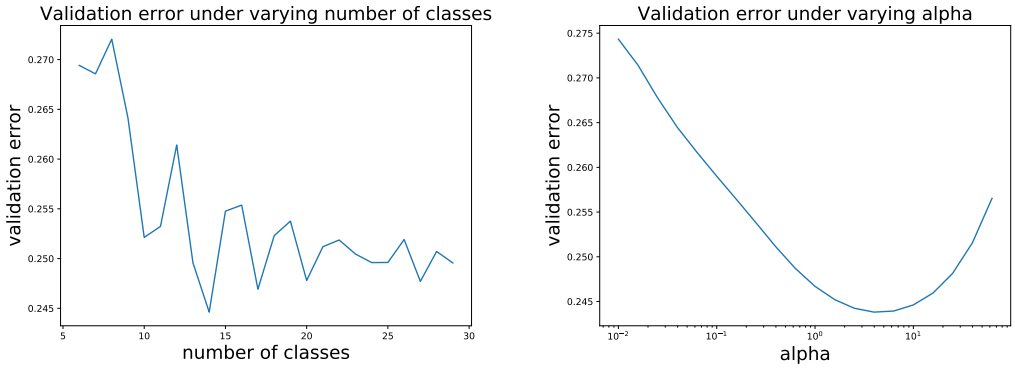
6 What next?

In section 5, we see that fitting the neural network jointly gives lower training error, yet higher validation error than fitting separately does. This suggests that fitting the neural network jointly somehow overfits the data, and thus simply adding more hidden layers to the network may not produce a better result. It's however sensible to investigate deeper on the method that fits the neural network separately, as it performs the best in terms of validation error among the methods tried so far. On the other hand, there are hyperparameters in the model which may not be optimal, meaning there is space for improvement.

In section 4, we trained a model which classifies the input into 10 classes in the first step. It's worth experimenting on tuning the number of classes to see if there is a better setting. In particular, we do a grid search from 6 to 40 with search interval 1, the coefficient of L2 penalty stays the same for consistency.

```
alpha = 10
K_list=[k for k in range(6,30,1)]
e_val=[]
for K in K_list:
    mx = np.max(y_train); mn = np.min(y_train); hh = (mx-mn)/(K+1)
    thresholds = np.linspace(mn+hh, mx-hh, num=K, endpoint=True)
    V = np.zeros((X_train.shape[1],K))
    V_b = np.zeros(K)
    for kk in range(K):
        labels = y_train > thresholds[kk]
        V[:,kk], V_b[kk] = fit_logreg_gradopt(X_train, labels, alpha)
    X_train_class = 1/(1 + np.exp(-(X_train.dot(V)+V_b)))
    X_val_class = 1/(1 + np.exp(-(X_val.dot(V)+V_b)))

    ww, bb = fit_linreg(X_train_class, y_train, alpha)
    e_val.append(rtmsq_error(ww, bb, X_val_class, y_val))
```



(a) Tuning number of classes

(b) Tuning coefficient of regularization

Figure 3: Validation error under different settings of hyperparameters

Figure 3(a) is a plot of the validation errors against number of classes. The lowest validation error is 0.2446, achieved by setting number of classes to 14. We can go further tuning the coefficient of regularization using the similar procedure. In particular, we set the number of classes to 14, and search for the optimal alpha in logarithm scale. The search grid is generated by the code below:

```
alpha_list=10**np.arange(-2,2,0.2)
```

Shown in figure 3(b) is the plot of validation errors against alpha. The lowest validation error is 0.2438, achieved by setting alpha to $10^{0.6} \approx 4$. This is lower than the best model in the previous sections, though not by very much (roughly 0.01). Finally we evaluate this model using the test set, the root mean square error is 0.2863.