1. AdaBoost Update

   Let $K_m(x_i)$ be the prediction made by the weak classifier at iteration m, then the overall prediction at iteration m can be denoted as $C_m(x) = \sum_{i=1}^{m} \alpha_m K_m(x) = C_{m-1}(x) + \alpha_m K_m(x)$.

   We can use the exponential loss function of Adaboost to derive the α.

   The loss function is: $E_m = \sum_{i=1}^{N} e^{-y_i C_m(x_i)}$, where y_i is the ground truth label of x_i.

   Substitute $C_m(x)$ into E_m, we can get: $E_m = \sum_{i=1}^{N} e^{-y_i(C_{m-1}(x_i)+\alpha_m K_m(x_i))}$.

   Since we can write the weight of a training sample i at m iteration like this: $w_i^{(m)} = e^{-y_i C_{m-1}(x_i)}$, substitute it into E_m, we can get: $E_m = \sum_{i=1}^{N} w_i^{(m)} e^{-y_i(\alpha_m K_m(x_i))}$.

   Since the labels for Adaboost are 1 and -1, then $y_i * K_m(x_i) = 1 \text{ or } -1$.

   We can write E_m as $\mathrm{E_m} = \sum_{y_i \neq K_m(x_i)} w_i^{(m)} e^{\alpha_m} + \sum_{y_i = K_m(x_i)} w_i^{(m)} e^{-\alpha_m}$.

   Since we want to minimize this loss function, we take the derivative with respect to α_m and let it equal to zero: $-\alpha_m e^{-\alpha_m} \sum_{y_i = K_m(x_i)} w_i^{(m)} + \alpha_m e^{\alpha_m} \sum_{y_i \neq K_m(x_i)} w_i^{(m)} = 0$.

   Solve the above equation for α_m: $e^{2\alpha_m} = \frac{\sum_{y_i = K_m(x_i)} w_i^{(m)}}{\sum_{y_i \neq K_m(x_i)} w_i^{(m)}}$.

   Now, let $W = \sum_{y_i \neq K_m(x_i)} w_i^{(m)}$ and $T = \sum_{i=1}^{N} w_i^{(m)}$. We can also have $T - W = \sum_{y_i = K_m(x_i)} w_i^{(m)}$.

   Therefore, $e^{2\alpha_m} = \frac{T-W}{W} = \frac{1-W/T}{W/T}$.

   Since $err\_m = W/T$, $e^{2\alpha_m} = \frac{1-err_m}{err_m}$.

   And $\alpha_m = \frac{1}{2}\log\left(\frac{1-err_m}{err_m}\right)$

2. Predicting Loan Defaults with Neural Networks
   (a) Similar to Homework 3, I split the data into a train set and a test set with a train: test ratio of 80%: 20%. I also continued to use the default 5-fold cross validation in the grid search.
   (b) Similar to Homework 3, I transfer all categorical data into numerical data. Then, I use Spearman correlation to compute the correlation matrix of the features and discard one feature from any pair of features with correlation score higher than 0.7. I also compute the correlation criteria of each feature with the label and select the top 10 features. Finally, I standardize the features to zero mean and unit variance. I decide to preprocess data this way because neural networks can only take in numerical data and tend to perform better when the input features are on the same scale. In addition, feature selection can help reduce computational complexity and prevent overfitting.
   (d) My search space is as follow:
   hiddenparams = [(16,), (32,), (64,), (32, 16), (64, 32), (128, 64), (64, 32, 16), (128, 64, 32)],
   actparams = ['logistic', 'tanh', 'relu'],
   alphaparams = [0.01, 0.1, 1, 10, 100]
   The optimal parameters are opt_hidden: (64, 32), opt_activation: logistic, and opt_alpha: 0.0001. (performance: 0.833672403508946)
   (e) The AUC, F1, and F2 scores of the neural model and decision tree with their best hyperparameters respectively are shown in *Table 1*. Both models are trained and tested on the same dataset which is preprocessed as described in 2(a) and 2(b).

Table 1 Performances of MLP and Decision Tree with Best Hyperparameters on Loan Default Prediction

model	AUC	F1	F2	train_time
MLP	0.7714606120191717	0.694981	0.6139154160982266	3.034517526626587
Decision Tree	0.7911392405063291	0.736	0.6353591160220995	0.006985

   (f) As shown in *Table 1,* compared to the decision tree (DT), the multilayer perceptron (MLP) has lower AUC, F1, and F2 scores. The MLP also takes significant more time to train than DT does, meaning that the neural network has a higher computational complexity. In addition, the MLP is much harder to tune than the DT, because it has 3 parameters to tune and the decision tree only has 2. Moreover, the hidden_layer_sizes of MLP needs to specify both number of layers and number of neurons in each layer. All of these characteristics of MLP require us to test on significantly more sets of parameters than we do for DT.

3. Stochastic Gradient Tree Boosting to Predict Appliance Energy Usage

(d) Figure 1 shows the RMSE of the Stochastic Gradient Tree Boosting (SGTB) with q=1, different nu and n_iter. The optimal parameters for q=1 are nu=0.1, n_iter=10. From the figure, we can see that when nu is fixed, larger n_iter actually causes RMSE to increase.

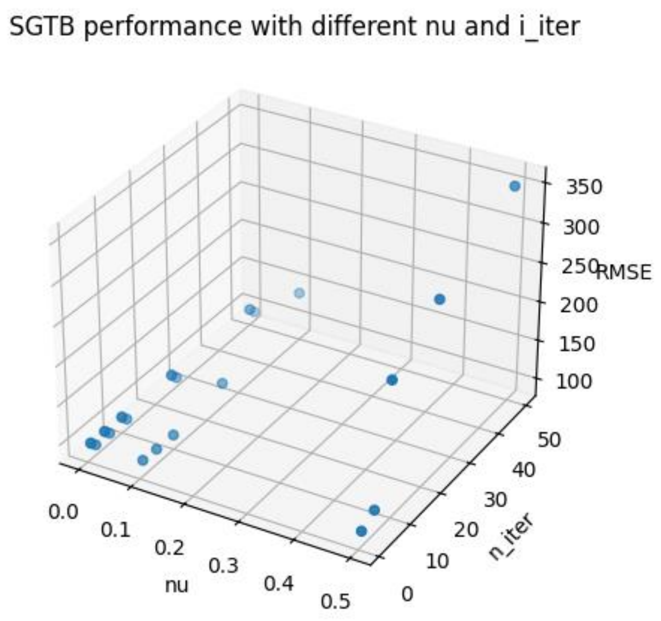

Figure 1

(f) The optimal parameters are nIter=25, nu=0.1, q=0.9. All the results are shown in Table 2.

Table 2 Performances of SGTB with Different Hyperparameters

nIter	nu	q	RMSE
1	0.1	0.6	107.3224439445524
1	0.1	0.7	107.01771552920191
1	0.1	0.8	106.9358
1	0.1	0.9	106.87120893714462
1	0.5	0.6	104.50163709783604
1	0.5	0.7	103.22040370426707
1	0.5	0.8	103.46867117619634
1	0.5	0.9	102.98670784873453
1	0.01	0.6	108.19067684314203
1	0.01	0.7	108.20061146708414
1	0.01	0.8	108.19974085580392
1	0.01	0.9	108.19674856205631
1	0.001	0.6	108.33829576340547
1	0.001	0.7	108.33758748002711
1	0.001	0.8	108.33583105249927
1	0.001	0.9	108.33710347768178
5	0.1	0.6	104.23950345017033

5	0.1	0.7	103.88842465238424
5	0.1	0.8	104.76293920742958
5	0.1	0.9	104.32017910989127
5	0.5	0.6	107.24066581054512
5	0.5	0.7	103.9974619140784
5	0.5	0.8	106.4813304942727
5	0.5	0.9	106.47579364980056
5	0.01	0.6	107.65211101894343
5	0.01	0.7	107.67636009827632
5	0.01	0.8	107.64562999302689
5	0.01	0.9	107.6149660652017
5	0.001	0.6	108.29368496035988
5	0.001	0.7	108.28734424191937
5	0.001	0.8	108.27914574915296
5	0.001	0.9	108.27643065462203
10	0.1	0.6	104.04794734112741
10	0.1	0.7	103.64501544082634
10	0.1	0.8	103.33703107005071
10	0.1	0.9	103.33460764112792
10	0.5	0.6	107.96016587848052
10	0.5	0.7	110.89505000460296
10	0.5	0.8	115.11308472474988
10	0.5	0.9	108.57740345479104
10	0.01	0.6	107.13107628101318
10	0.01	0.7	107.04836362986524
10	0.01	0.8	106.9896188645138
10	0.01	0.9	106.9621
10	0.001	0.6	108.21905721770963
10	0.001	0.7	108.20613196286486
10	0.001	0.8	108.20469429959367
10	0.001	0.9	108.19879228384964
25	0.1	0.6	103.59519102932109
25	0.1	0.7	103.99194047515523
25	0.1	0.8	103.84449896326933
25	0.1	0.9	102.1868805123482
25	0.5	0.6	129.2429444694091
25	0.5	0.7	142.05079610372675
25	0.5	0.8	119.39554050315425
25	0.5	0.9	120.1071721334998
25	0.01	0.6	105.75707287555649
25	0.01	0.7	105.6991071138976
25	0.01	0.8	105.63757621861514
25	0.01	0.9	105.45166234144605
25	0.001	0.6	108.00570224892901

25	0.001	0.7	107.99983191139624
25	0.001	0.8	107.98496302265819
25	0.001	0.9	107.98295731992107
50	0.1	0.6	107.21498329173305
50	0.1	0.7	104.62454280516815
50	0.1	0.8	104.78804455107809
50	0.1	0.9	102.99597917750961
50	0.5	0.6	162.5734528574343
50	0.5	0.7	142.60317650997703
50	0.5	0.8	139.88345051972414
50	0.5	0.9	139.60121014761904
50	0.01	0.6	104.6429858990817
50	0.01	0.7	104.3813130857529
50	0.01	0.8	104.25854147600887
50	0.01	0.9	104.19958558056842
50	0.001	0.6	107.69057884408603
50	0.001	0.7	107.66829896147927
50	0.001	0.8	107.64407914701835
50	0.001	0.9	107.62347909818156

(g) From Homework 1, the optimal performance of the linear regression model is RMSE=454.148, and the optimal performance of the ElasticNet is RMSE=283.1666. In comparison, the optimal performance of the SGTB model is RMSE=150.449, which is better than both models from Homework 1.

(h) Training both the SGTB and the gradient tree boosting (GTB) on the train + val set with nIter=10 and nu=0.1. SGTB has RMSE=113.258 and train_time=0.681. In compariSson, GTB has RMSE=111.338 and train_time=0.740. The performances and training times are quite similar. This might be caused by setting q=0.9 for SGTB, which would lead to training on similar samples sizes of data for both models in each iteration. However, GTB still has a higher RMSE and longer train_time than SGTB does. This makes sense because GTB uses all the training data in each iteration and thus requires more computation time and can fit the data better. In addition, hyperparameter tuning for SGTB is harder than that for GTB, because besides nIter and nu, SGTB also needs to search for optimal q.