CS 534: Homework #3

Submission Instructions: The homework is due on Oct 21st at 11:59 PM ET on Gradescope. A part of your homework will be automatically graded by a Python autograder. The autograder will support Python 3.10. Additional packages and their versions can be found in the requirements.txt. Please be aware that the use of other packages and/or versions outside of those in the file may cause your homework to fail some test cases due to incompatible method calls or the inability to import the module. We have split homework 1 into 2 parts on Gradescope, the autograded portion and the written answer portion. If either of the two parts is late, then your homework is late.

1. Upload PDF to HW3-Written Assignment: Create a single high-quality PDF with your solutions to the non-coding problems. The solutions must be typed (e.g., Word, Google Docs, or LaTeX) and each problem appropriately tagged on Gradescope. If we must search through your entire PDF for each problem, you may lose points! Note that you must submit the code used to generate your results in the Code part of the assignment, otherwise you may not get any points for the results.

2. Submit code to the HW3-Code Assignment: Your submitted code must contain the following files:

q1.py, q2.py, 'README.txt'.

You must submit ALL files you used to generate the results but the autograder will only copy these files when running the test cases so make sure they are self-contained (i.e., capable of running standalone). Make sure you always upload ALL of these files when you (re)submit. The README.txt file must contain a signed honor statement that contains the following words:

```
/* THIS CODE IS MY OWN WORK, IT WAS WRITTEN WITHOUT
CONSULTING CODE WRITTEN BY OTHER STUDENTS
OR LARGE LANGUAGE MODELS LIKE CHATGPT.
Your_Name_Here */
I collaborated with the following classmates for this homework:
<names of classmates>
```

1. **Energy appliance regression** (50 pts, written + code):

We again use the Energy dataset (energydata.zip) from Homework #1. As a reminder, each sample contains measurements of temperature and humidity sensors from a wireless network, weather from a nearby airport station, and the recorded energy use of lighting fixtures to predict the energy consumption of appliances in a low energy house. Put all code into q1.py. There are 2 classes: **FeatureSelection** and **Regression**. Complete implementation of the following class methods. You may use helper methods from the *sklearn.feature_selection* module and regressors from *sklearn.linear_model***.**

(a) **Code: FeatureSelection.rank_correlation(x, y)** that takes in a numpy 2d array, **x**, a numpy 1d array, **y**, and returns the rank of the features in x based on Pearson correlation of each individual feature to the target value. The function should return a numpy array in descending order of the most correlated feature column (i.e., [3, 1, 0, 2] would mean that the 4th feature has the highest correlation to ***y*** followed by 2nd, then 1st, and lastly 3rd feature.)

(b) **Code: FeatureSelection.Lasso(x, y)** that takes in a numpy 2d array, **x**, a numpy 1d array, **y**, and returns the rank of the features in x based on coefficients of Lasso regression (L1 regularization). The function should return a numpy array in descending order of the absolute magnitude of Lasso regression coefficient. *do not include features with zero Lasso coefficient*. For example, for 4-features input X, if Lasso coefficients are [2.3, -4.5, 0, 7.6], we should return [3, 1, 0], to mean that the 4th feature has the highest coefficient followed by 2nd, then 1st, and the 3rd feature is dropped. Note that the returned array is likely to be shorter than the original dimension of X.

(c) **Code: FeatureSelection**.**stepwise(x, y)** that takes in a numpy 2d array, **x**, a numpy 1d array, **y**, and returns the rank of the features in **x** based by greedy adding one remaining feature at a time to the set. To do this, we try adding one remaining feature at a time to the set, train a *linear regressor*, and select next feature to add based on the highest decrease in RMSE. Stop the process when RMSE does not decrease. For simplicity, we do this process on the training set X, but feel free to use cross validation for this step. See slides for details or read about reference implementation *SequentialFeatureSelector.* Return ranking of features in descending order of importance. Note that the returned array is likely to be shorter than the original dimension of X.

(d) Written: report up to 10 first features selected by each method 1a,1b, and 1c in a table. Comment on overlap and differences in the resulting feature sets.

(e) Code: **Regression.Ridge(train_x, train_y, test_x, test_y):** Implement Ridge regression and returns prediction on test set similar to HW1. You may use sklearn for the regressor implementation.

(f) Written: use your code in 1e to train and test Ridge regression on the whole feature set ("No selection"), and on top 10 features selected in 1a, 1b, 1c. Report in a table the RMSE and R^2 of the Ridge regressor on the *validation dataset and test dataset*. Comment on differences/similarities of performance of Ridge regression under the different feature selection methods.

(g) Code: **Regression.DecisionTreeRegressor (train_x, train_y, test_x, test_y, max_depth, min_items):** Implement Regression Tree training and prediction, and return prediction on validation and test sets similar to HW1. You may use sklearn for the underlying regressor tree implementation.

(h) Written: tune the hyperpameters of your regression tree in 1g on whole feature set by varying the **max_depth** and **min_items** and selecting the best configuration based on the *validation set*. Report the best configuration and RMSE and R^2 obtained on *validation set.*

(i) Written: Train the regressor using best configuration on full dataset as training (train'=train+validation), and report RMSE and R^2 on test dataset. Compare performance of your tree regressor to the best performance of Ridge regression in **1f**.

(j) Written: visualize top 3 levels in your decision tree in 1h and 1i. Compare the top-level features used by your tree, with the features selected in 1d. Comment on overlap/order of importance of features in the two methods.

2. **Code+Written** (50 pts) Spam classification using Naïve Bayes, Random Forest, and GBDT.

We will use the same email spam dataset from HW2 the email spam dataset, which contains 4601 e-mail messages that have been split into 3000 training (spam.train.dat) and 1601 test emails (spam.test.dat). 57 features have been already extracted for you, with a binary label in the last column.

All the specified functions should be in the file 'q2.py'.

(a) (Code) Write a Python function **eval_randomforest(trainx, trainy, testx, testy, num_trees, max_depth, min_items)** that fits a Random Forest model to the training data. You can use **sklearn** module for this part. The function should accept as input numpy 2d arrays, and return a dictionary containing the accuracy and AUC for the training and test sets and the predicted probabilities for the test set: return {"train-acc": train_acc, "train-auc": train_auc, "test-acc": test_acc, "test-auc": test_auc, "test-prob": test_prob}. The values for accuracy and AUC should be scalar numeric values, while test-prob should be a numpy 1-d array with the predicted probability for the positive class 1, for the test. Same format as HW2.

(b) (Written) Tune RandomForest for the Spam classification problem by optimizing the hyperparameters **num_trees, max_depth, min_items** based on cross validation over the training set only. Report best configuration, and average AUC and Accuracy across validation folds.

(c) (Written) Train Random forest on all training data with using best hyperparameters identified in 2b, and report AUC, F1, and Accuracy on the test set. Compare in a table to the performance to NB and LR from Homework 2.

(d) Written: Report top 10 most important features for NB and RF (describe your criteria chosen for "most important" in each case).

(e) (Code) Write a Python function **eval_gbdt(trainx, trainy, testx, testy, num_estimators, learning_rate)** that fits a **GradientBoostingClassifier** model to the training data**.** You can use **sklearn** module for this part. The function should accept as input numpy 2d arrays, and return a dictionary containing the accuracy and AUC for the training and test sets and the predicted probabilities for the test set: return {"train-acc": train_acc, "train-auc": train_auc, "test-acc": test_acc, "test-auc": test_auc, "test-prob": test_prob}. The values for accuracy and AUC should be scalar numeric values, while test-prob should be a numpy 1-d array with the predicted probability for the positive class 1, for the test. Same format as HW2.

(f) Written Tune GBDT for the Spam classification problem by optimizing the hyperparameters **num_estimators** and **learning_rate** at least (you may choose to tune others). Use cross validation over the training set only. Report the best configuration, and the average AUC, F1, and Accuracy across validation folds.

(g) (Written) Train GBDT on all training data with your best hyperparameters, and report AUC, F1, and Accuracy on the test set. Compare in a table to the performance of RF and NB. Comment on relative improvements / performance for this problem by the three different models.