CS 534: Machine Learning
Emory University
Fall2024
Prof. Eugene Agichtein

Homework # 3

Mitchell Scott
(mtscot4)

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, $\mathbf{x}$, a numpy 1d array, $\mathbf{y}$, and returns the rank of the features in $\mathbf{x}$ 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 $\mathbf{y}$ followed by 2nd, then 1st , and lastly 3rd feature.)
   Solution. Implimented in `q1.py`.

   (b) **Code:** `FeatureSelection.Lasso(x, y)` that takes in a numpy 2d array, $\mathbf{x}$, a numpy 1d array, $\mathbf{y}$, and returns the rank of the features in $\mathbf{x}$ based on coefficients of Lasso regression (ℓ^1 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 $\mathbf{X}$, 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 $\mathbf{X}$.
   Solution. Implimented in `q1.py`.

   (c) **Code:** `FeatureSelection.stepwise(x, y)` that takes in a numpy 2d array, $\mathbf{x}$, a numpy 1d array, $\mathbf{y}$, and returns the rank of the features in $\mathbf{x}$ 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 $\mathbf{X}$, 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 $\mathbf{X}$.

Selection Method	Train RMSE	Train r2	Test RMSE	Test r2
Full	**0.94963**	**0.18675**	**0.95906**	**-0.21560**
Correlation	0.49582	0.71419	1.41650	-8.79686
Lasso	0.56684	0.88240	0.97312	-1.49702
Stepwise	0.78201	0.59070	1.05242	-2.13966

Table 1: By changing the features selected, we see drastically different training and test accuracies.

Solution. Implimented in `q1.py`.

(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.

Solution. For the `rank_correlation` method, we arrived at $[24, 23, 8, 10, 9, \cdots 20, 6, 17, 22, 2]$. For the `lasso` method, we arrived at, $[12, 22, 21, 20, 19, 17, 15, \cdots 14, 13, 24]$. Lastly, For the `stepwise` method, we arrived at, $[12, 22, 20, 19, 17, \cdots 15, 14, 10, 24, 7]$. We see that `lasso` and `stepwise` give rather similar results with the first 7 out of 8 entries being the same. Additionally, we see 24 represented close to the bottom in both methods as well. Comparing either `lasso` or `stepwise` with `rank_correlation`, we see that the results are not all that similar. Of course all methods have 24, 20, 17, and 22, but whereas, `lasso` starts with 22,20, and 17, `rank_correlation` has those entries at the end. Conversely, `stepwise` has 24 as one of the last, but `rank_correlation` has it first.

Additionally, while `lasso` and `stepwise` have very similar results, `stepwise` took noticibly longer to compute.

(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

Solution. Implimented in `q1.py`.

(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.

Solution. As we can see in Tab 1, by training on the full data set, we get considerably better training and test RMSE and R^2.

(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.

Solution. Implimented in `q1.py`.

(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.

Parameters (`max_depth, min_items`)	Train RMSE	Train r2	Test RMSE	Test r2
(2,1)	0.97108	0.11075	0.92638	-0.05817
(2,3)	0.97108	0.11075	0.92638	-0.05817
(2,5)	**0.97108**	**0.11075**	**0.92638**	**-0.05817**
(5,1)	0.92628	0.26386	1.01166	-0.50499
(5,3)	0.92536	0.26678	1.01166	-0.50499
(5,5)	0.92620	0.26411	1.01166	-0.50499
(8,1)	0.83043	0.52443	1.07061	-0.88766
(8,3)	0.83169	0.52155	1.02947	-0.61385
(8,5)	0.84739	0.49882	1.02207	-0.56794

Table 2: Tuning the hyperparameters for a Decision Tree Regressor. Bolded is best.

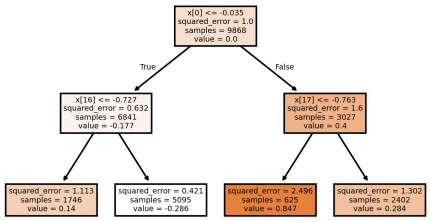

Figure 1: Decision Tree H

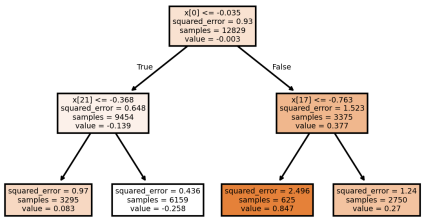

Figure 2: Decision Tree I

Solution. By iterating over `max_depth` = [2,5,8], `min_items` = [1,3,5], we were able to see the dependence on these parameters in terms of training the regression tree. (2,5) won. Alll results are seen in Tab 2.

(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.

Solution. Using the full data set, we computed, the following dictionary:

$$\{\texttt{'test-rms'} : 0.90088, \quad \texttt{'test-r2'} : 0.05363.\}$$

We are able to see that our test RMSE is lower and our R^2 value is more positive but smaller in magnitude. I feel that a positive correlation is better, so a success on both counts.

(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.

Solution. We see the decision trees for 1h and 1i in Fig 1 and Fig. 2, respectively. 17 appears in both, which is good because feature 17 was mentioned in every single feature selection method.16 and 21 were mentioned once each and were mentioned in just one of the feature selection methods.

2. **Spam classification using Naïve Bayes, Random Forest, and GBDT** (50 pts, written + code): 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.

Solution. Implimented in `q3.py`.

(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.

Solution. By going from 1 to 16 estimators, 1 to 15 depth and 1 to 15 items per leaf, we found that the best was 16 estimators, max depth of 15 and 9 min items. The resulted in the following corss validated data.

'train-acc': 0.95043, 'train-auc': 0.94303, 'val-acc': 0.93263, 'val-auc': 0.92469

(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.

Solution. Using the best Params (num=16, depth=15, items =9) on the full data set, we found

'train-acc': 0.94932, 'train-auc': 0.94193, 'test-acc': 0.9275, 'test-auc': 0.91858, 'f1-score': 0.90794, 'test-prob': array([0.99657635, 0.00342365])

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

Solution. Not knowing how to approach this problem, I consulted sklearn. They have a feature selection tool which finds the features that are computed as the mean and standard deviation of accumulation of the impurity decrease within each tree. It produced the following plot In Fig. 3, we see that the features at the beginning and end tend to be the most important as they have the highest values.

(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":`

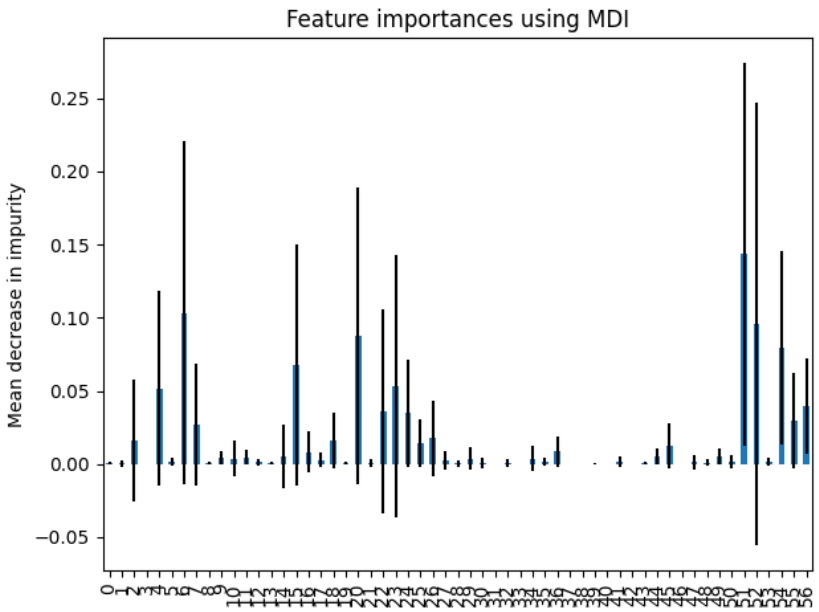

Figure 3: Features which reduce the standard deviation the most by adding them

`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.

Solution. Implimented in `q3.py`.

(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.

Solution. We were able to iterate through many different estimators (1-15) and many different learning rates (0.1,0.01,0.001,0.0001,0.00001). We found the best was 12 estimatros and 0.0001 learing rate. The following was the results 'train-acc': 0.94932, 'train-auc': 0.94193, 'test-acc': 0.9275, 'test-auc': 0.91858, 'f1-score': 0.90794, 'test-prob': array([0.99657635, 0.00342365]). This was better than NB on the last homework.

(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.

Solution. Best Params: 'train-acc': 0.6153333333333333, 'train-auc': 0.5, 'test-acc': 0.5883822610868208, 'test-auc': 0.5, 'f1-score': 0.0, 'test-prob': array([0.61536766, 0.38463234]) This is much worse than NB. I don't know what happened but by training on the whole data set all at once, it didn't seem to work well.

Acknowledgements

I would like to acknowledge that I attended both TA Swati's and TA Ziyang's office hours over the past week.