CS 534: Machine Learning
Emory University
Fall 2024
Prof. Eugene Agichtein

Homework # 2

Mitchell Scott
(mtscot4)

1. **Written(10 pts) Bias-Variance Trade-off of LASSO** While it is hard to write the explicit formula for the bias and variance of using LASSO, we can quantify the expected general trend. Make sure you justify the answers to the following questions for full points:

   (a) **(Written) What is the general trend of the bias as λ increases?**

   Solution. The general trend for Lasso is as λ increases, the bias increases. As λ increases, that means that if there are more features, the $\|\beta\|_1$ will increase, and that is scaled by λ, which is a very large number. This incentivises the model to include as few features as possible in very small amount. Using less features than needed, is the classic underfitting problem, so we have high bias.

   (b) **(Written) What about the general trend of the variance as λ increases?**

   Solution. As λ increases, the variance decreases. This makes sense as we have a simpler model with less parameter/ features that we are dealing with. This means that we cannot possible overfit, which is common with high variance situations. Since the model is simpler, then it is less data or problem dependent and can be used for other data sets rather easily.

   (c) **(Written) What is the bias at $\lambda = 0$?**

   Solution. When $\lambda = 0$, we must recall that the regularization term $+\lambda\|\beta\|_1$ becomes zero, so the loss function we are fitting is simpily the ordinary least squares problem, or Linear Regression model.

   (d) **(Written) What about the variance at $\lambda = \infty$?**

   Solution. With λ this high, that means we have a very biased model with very low variance. If there were any feature at all, then the $\|\beta\|_1$ would be positive, and the loss function would be infinite. This means we cannot afford to imcorporate any features in our model, so we simply have the constant intercept as our model.

2. **Code+Written(40 pts) Spam classification using Naive Bayes and Standard Logistic Regression**

   (a) **(Code) You will explore the effects of feature preprocessing and its impact on Naive Bayes and Standard (unregularized) logistic regression. Write the following functions to preprocess your data. You**

are free to use the `sklearn.preprocessing` module. Note that they functions should be independent of one another and should not build on each step/call each other. You should assume only the features are passed, in and not the target. These functions should accept numpy 2darray as input, return the preprocessed train and test set in numpy 2D array format (i.e., two return values).

i. **function `do_nothing(train, test)` that takes a train and test set and does no preprocessing.**

ii. **function `do_std(train, test)` that standardizes the columns so they all have mean 0 and unit variance. Note that you want to apply the transformation you learned on the training data to the test data. In other words, the test data may not have a mean of 0 and unit variance.**

iii. **function `do_log(train, test)` that transforms the features using $\log(x_{ij} + 0.1)$ or a smoothed version of the natural logarithm.**

iv. **function `do_bin(train, test)` that binarize the features using $\mathbb{I}_{(x_{ij}>0)}$. (Note that $\mathbb{I}$ denotes the indicator function). In other words, if the feature has a positive value, the new feature is a 1, otherwise, the value a 0.**

Solution. Implimented in '`q2.py`'.

(b) **(Code) Write a Python function `eval_nb(trainx, trainy, testx, testy)` that fits a Naive Bayes model to the training. You can use `sklearn.naive_bayes` 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.

Solution. Implimented in '`q2.py`'.

(c) **(Written) Fit a Naive Bayes model to each of the four preprocessing steps above using the code in 2b. Each preprocessing should be performed independently (i.e., use each of the functions you created in 2a on the original dataset). Report the accuracy rate of and AUC of NB on the training and test sets across the 4 preprocessing steps in a table.**

Solution. We compare the different preprocessing methods based on training and test accuracy and area under the ROC curve (auc) in Tab. 1. As we see from the bolded numbers, we have the largest (either highest accuracy or highest auc, which is preferable) number in the logorithmic scaling, so it appears on these four metrics, `do_log` is the best preprocessor for the Naive Bayes model using Spam classificiation.

(d) **(Code) Write a Python function `eval_lr(trainx, trainy, testx, testy)` that fits a ordinary (no regularization) logistic regression model. The**

	'train-acc'	'train-auc'	'test-acc'	'test-auc'
do_nothing	0.82561	0.85096	0.81750	0.83712
do_std	0.81627	0.84305	0.81063	0.83127
do_log	**0.83228**	**0.85573**	**0.81875**	**0.83795**
do_bin	0.79860	0.82739	0.80063	0.82277

Figure 1: Comparing the different preprocessing methods for Naive Bayes on the Spam Classification task. Bolded means the result is the highest of all methods in that metric.

function should return a dictionary containing theaccuracy and AUC for the training and test sets and the predicted probabilities for the test:

{"train-acc": train_acc, "train-auc": train_auc, "test-acc": test_acc, "test- auc": test_auc, "test-prob": test_prob}.
Note that the values for accuracy and AUC should be scalar numeric values, while test-prob should either be a numpy 1-d array with the predicted probability of positive class 1 for the test set. The output will be the same format as 2b.

Solution. Implimented in 'q2.py'.

(e) **(Written) Fit ordinary (no regularization) logistic regression model with each of the four preprocessing steps above using the code in 2d. Report the accuracy rate and AUC on the training and test sets for the 4 preprocessing steps in a table**

Solution. We compare the different preprocessing methods based on training and test accuracy and area under the ROC curve (auc) in Tab. 2. As we see from the bolded numbers, we have the largest number in the logorithmic scaling, so it appears on these four metrics, do_log is the best preprocessor for the Logistic Regression model using Spam classificiation. Additionally, it is noted

	'train-acc'	'train-auc'	'test-acc'	'test-auc'
do_nothing	0.93398	0.92817	0.92063	0.91273
do_std	0.93464	0.92871	0.92063	0.91274
do_log	**0.94431**	**0.94111**	**0.93313**	**0.92677**
do_bin	0.93898	0.93417	0.92500	0.91691

Figure 2: Comparing the different preprocessing methods for Logistic Regression on the Spam Classification task. Bolded means the result is the highest of all methods in that metric.

that the do nothing preconditioning method resulted in a 'did not converge' error using LBFGS up to 5000 iterations, so it was rerun with Newton-CSG as the iterative solver, where there was convergence for all of the preprocessing steps. While the exact accuracies are slightly different, they are the same as the LBFGS method for up to 4 decimal places, and the overall trends of logarithmic preprocessing still reigns supreme. Lastly, it is imperative to note that Logistic Regression did between 10 and 15 % better than Naive Bayes for the same data set.

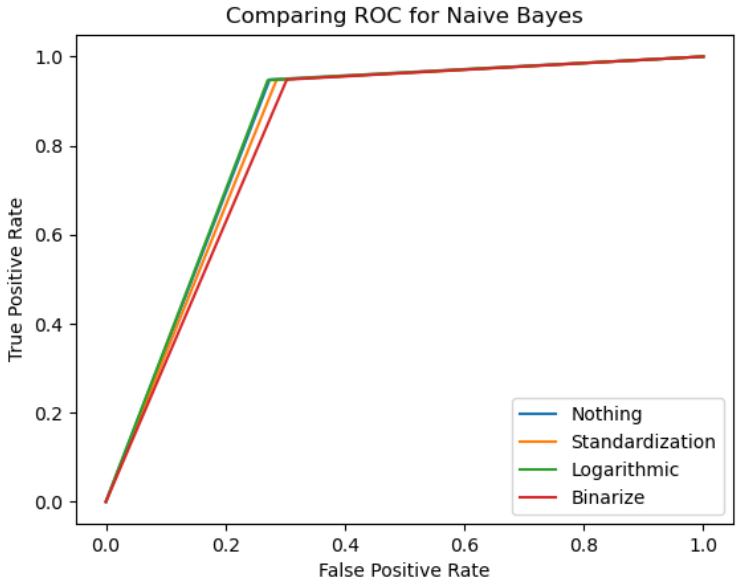

Figure 3: Comparing the ROC curves for different preprocessing features for Naive Bayes.

(f) **(Written) Plot the receiver operating characteristic (ROC) curves for the test data. You should generate 3 plots:**

- **One plot containing the 4 Naive Bayes model curves representing each of the preprocessing steps.**
- **One plot containing the 4 logistic regression model curves representing each of the preprocessing steps.**
- **One plot containing the best Naive Bayes model and the best logistic regression model curve.**

Solution. There does appear to be something weird with these ROC curves as there is only three points. I feel that I have done something wrong there as they should be more smooth.

(g) **(Written) Given your results in 2c, 2e, and 2f, discuss how the preprocessing affects the models (Logistic and Naive Bayes) with regards to ROC, AUC, and accuracy. Also, comment on how Naive Bayes performance compares with logistic regression**

Solution. We notice that from the plots and tables that the best preprocessing method is logarithmic, then binary, then standarization barely surpassing doing nothing. We also observe that the logistic regression is about 10-15% more accurate than naive Bayes.

3. **(50 pts) Exploring Model Selection Strategies for Logistic Regression with Regularization We will be using the SPAM dataset from the previous part for this problem. You can preprocess the data however you see fit, either based on the results of the previous problem or by introducing another preprocessing method. The only requirement is that it is consistent throughout the rest of this problem. For this problem, you are not allowed to use the `sklearn.model_selection` module. All the specified functions should be in the file `'q3.py'`.**

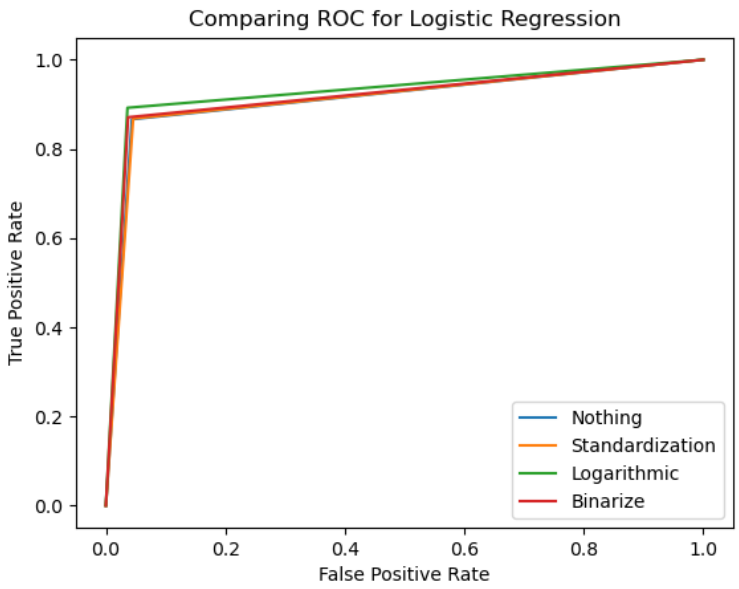

Figure 4: Comparing the ROC curves for different preprocessing features for Logistic Regression.

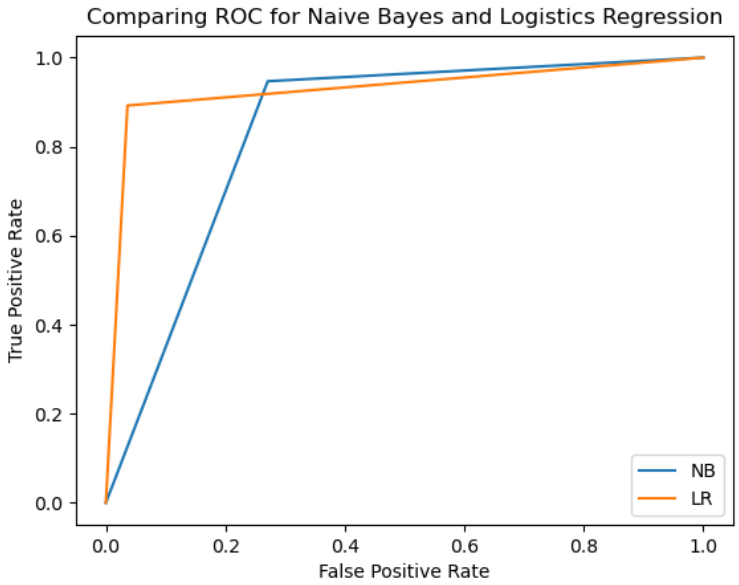

Figure 5: Using the best preprocessing function (logarithmic), we then compare how Naive Bayes and Logistic Regression perform on this dara by the ROC curve metric.

(a) **(Written) How did you preprocess the data for this problem? Why?**

Solution. Based on the results up above, the data was preprocessed using the logarithmic scaling. This was preformed exactly as it was in question 2. We initially separated the train and test data from the different .dat files. Then we did log scaling on the train and test x values, then we concatenated the `trainx, testx` and `trainy,testy`.

(b) **(Code) Implement the Python function `generate_train_val(x, y, valsize)` that given the validation size splits the data randomly into train and validation splits. The function should return a dictionary** `return {"train-x": tr_x, "train-y": tr_y, "val-x": ts_x, "val-y": ts_y}`.
The values for 'train-x' and 'val-x' are expected to be numpy 2d arrays of the same dimension as x, and split into the associated training and validation features. The values for 'train-y' and 'val-y' are expected to be a subset of y split accordingly. Note that each time this function is run, the splits could be different.

Solution. Implimented in 'q3.py'.

(c) **(ridgeCode) Implement the Python function `generate_kfold(x, y, k)` that given the k, will split the data into k-folds. The function should return a single numpy 1-d array, containing the k that each item index it belongs to (e.g., array([0,1,2,. . . ,2,1,1])) for $k = 3$, which indicates item 0 belongs to fold 0, and item 1 belongs to fold 1, etc.**

Solution. Implimented in 'q3.py'.

(d) **(Code) Implement the Python function`eval_holdout(x, y, valsize, logistic)` which takes in the input (e.g., 3000 training samples from spam.train.dat), the input labels, and the validation size and (1) uses 3b to split x and y into train and validation, and (2) evaluates the performance using the logistic regressionmodel passed in as . You can assume the logistic regression classifier will be created using `sklearn.linear_model.LogisticRegression` and initialized before passed to your method, so you can invoke it as usual. Your function should return a dictionary containing the accuracy and AUC for the training and validation sets using keys 'train-acc', 'train-auc', 'val-acc', 'val-auc'.**

Solution. Implimented in 'q3.py'.

(e) **(Code) Implement function `eval_kfold(x, y, k, logistic)` which takes in number of folds k, (1) uses 3c to split the data, and (2) evaluates the performance using the input classifier logistic instantiated as logistic regression model. You can assume the logistic regression classifier will be created using `sklearn.linear_model.LogisticRegression`. Your function should return a dictionary containing the accuracy and AUC for the training and validation sets using the following keys: 'train-acc', 'train-auc', 'val-acc', 'val-auc'. The accuracy and AUC for this part should be a averaged across the k folds.**

Solution. Implimented in 'q3.py'.

(f) **(Code) Implement the Python function `eval_mccv(x, y, valsize, s, logistic)` that takes in the validation size and the sample size s and uses the Monte Carlo cross-validation approach with s rounds (i.e., use the validation/hold-out technique from 3d, s times). The output should be the same format as 3e.**

Solution. Implimented in `'q3.py'`.

(g) **(Written) Fit Ridge and LASSO using the K-fold cross validation approach with $k = 5, 10$, and varying alpha (regularization weight). Report the best setting (combination of k and regularization weight alpha)**

Solution. First, it is important to know that when we are building these, the `C` parameter is the 'inverse of regularization strength' from the SciKit-Learn documentation. So as opposed to putting α as the regularization parameter, we have to set it up so that $C := \frac{1}{\alpha}$.

Since We don't know where to start, we use $C = 1$, which is the default for the Logistic Regression Model. Additionally, to make everything fairer, we set the solver to `'liblinear'`, as that is the first solver that works on both ℓ^1 and ℓ^2 regularization. Below are the results. We see in Tab. 6 that for Ridge

	`'train-acc'`	`'train-auc'`	`'test-acc'`	`'test-auc'`
Ridge, $k = 5$	0.94253	0.93799	0.93928,	0.93452
Ridge, $k = 10$	**0.94334**	**0.93902**	**0.94061**	**0.93618**
Lasso, $k = 5$	0.94515	**0.94110**	**0.94146**	**0.93695**
Lasso, $k = 10$	**0.94518**	0.94108	0.94027	0.93624

Figure 6: Comparing the variations of Logistic Regression regularization and k for $k-$folds on the Spam Classification task. Bolded means the result is the highest of all methods in that metric.

regression, $k = 10$ is better for every metric, but for Lasso regression $k = 5$ is better especially for the test data.

Lastly, to get varying α, we ran a loop for every α value from 0 to 1 up to three decimimal places. This information uses `test-acc` as the metric we are comparing. As we can see from Tab. 7, depending on the model and k value,

Model	`'test-acc'`	$\hat{\alpha}$
Ridge, $k = 5$	0.94278	$\hat{\alpha} = 0.248$
Ridge, $k = 10$	0.94293	$\hat{\alpha} = 0.530$
Lasso, $k = 5$	0.94260	$\hat{\alpha} = 0.271$
Lasso, $k = 10$	0.94382	$\hat{\alpha} = 0.992$

Figure 7: Up to 3 decimal places, these are the best $\hat{\alpha}$ values, which produce the highest `'test-acc'` in the respective models

we have very different $\hat{\alpha}$ values, but similar, high test-accuracy.

(h) **(Written) Fit Ridge and LASSO using the Monte Carlo Cross-validation (MCCV) approach with $s = 5, 10$ rounds and different valsize ratios,**

and varying alpha (regularization weight). Report the best setting (combination of valsize, s, alpha).

Solution. Just like above, we iterate through α values and then make the Logistic Regression model for both Ridge and Lasso with $C = \frac{1}{\alpha}$. We iterate through all of the values for alpha between 0 and 1 up to three decimal places. Additionally, we have the `valsize` parameter to iterate through as well, so I set the possible values to be 46, 92, 138, which corresponds to 10, 20, and 30 % of the total dataset ($n = 4601$). We see many interesting things in Tab. 8,

Model	`'test-acc'`	$\hat{\alpha}$	Test Data %
Ridge, $s = 5$	0.97826	$\hat{\alpha} = 0.950$	20%
Ridge, $s = 10$	0.96957	$\hat{\alpha} = 0.390$	20%
Lasso, $s = 5$	0.98261	$\hat{\alpha} = 0.569$	20%
Lasso, $s = 10$	0.97174	$\hat{\alpha} = 0.332$	20%

Figure 8: Up to 3 decimal places, these are the best $\hat{\alpha}$ values, which produce the highest `'test-acc'` in the respective models with what percent of the data was test data.

namely that when $s = 5$, the model is about 1% more accurate. Additionally, the 80/20 train/test split appears to universally be the best way to partitiion the data. However, I am slightly doubious about these results as we see almost 100% accuracies, which is making me question if we are possible mixing our test and training data and memorizing it.

(i) **(Written) Using the best parameters identified in 3g and 3h, re-train the regularized logistic regression models (Ridge, Lasso) using all the training data and report the performance on the test set in terms of AUC and accuracy in a table. You should have 4 models to compare: Ridge and Lasso trained using best parameters from k-fold and Ridge and Lasso using best parameters from MCCV. Compare how the model selection techniques compare to one another w.r.t. AUC and accuracy, predicted vs. actual test error, and computational complexity (running time).**

Model	`'train-acc'`	`'train-auc'`	`'test-acc'`	`'test-auc'`
Ridge, k-fold	0.94388	0.93969	0.94031	0.93583
Lasso, k-fold	0.94492	0.94089	0.94102	0.93719
Ridge, MCCV	0.94359	0.93933	0.93478	0.92793
Lasso, MCCV	0.94447	0.94947	0.92609	0.91980

Figure 9: Comparing the Optimal Parameters for k-fold and MCCV using Ridge and Lasso Logistic Regression.

Solution. We see from the Tab. 9, that Lasso is better than Ridge in both settings and k fold is better than the respective mccv value. In terms of running time, both models took about 1 second to run. Additionally k-fold has a better predicted vs actual test error.

Acknowledgements

I would like to acknowledge that I didn't work with any other CS 534, nor did I attend TA or Professor OH this week.