Archit Sakhadeo
Homework Assignment # 3
Due: Friday, November 22, 2019, 11:59 p.m.
Total marks: 100

Question 1. [60 MARKS]

In this question, you will implement several binary classifiers: naive Bayes, logistic regression and a neural network. An initial script in python has been given to you, called `script_classify.py`, and associated python files. You will be running on a physics dataset, with 8 features and 100,000 samples (called `susysubset`). The features are augmented to have a column of ones, in `dataloader.py` (not in the data file itself). Baseline algorithms, including random predictions and linear regression, are used to serve as sanity checks. We should be able to outperform random predictions for this binary classification dataset.

(a) [15 MARKS] Implement naive Bayes, assuming a Gaussian distribution on each of the features. Try including the columns of ones and not including the column of ones in the predictor. What happens? Explain why.

Solution: Implemented as Class **NaiveBayes** in the classalgorithms.py file.

If we ignore the column of ones, I get an average error of 25.63 with a standard deviation of 0.157 in the errors for my Naive Bayes classifier over 20 runs.

Including the column of ones leads to a variance of 0 over that specific feature. If we assume a Gaussian distribution over the features, then the probability of that specific feature given the class is undefined. This is because the probability for a Gaussian distribution requires dividing by variance which is zero for the feature with a column of ones. This gives a divide by zero error for all data samples. We can handle the situation of zero variance differently the way it is done in the **gaussian_pdf()** function in the utilities.py file by checking the difference between the value of the feature and mean value for that feature. If the difference is very small, we assign it a probability of 1 else we assign it a probability of 0. In this case, including the column of ones does not make any difference. The average error and standard deviation for including the column of ones is the same as the average error of 25.63 and standard deviation of 0.157 for not including the column of ones, over 20 runs.

Adding a constant feature of all ones to the data does not affect the output of the Naive Bayes Classifier, as it does not lead to any bias the way it does for Linear Regression. The feature with column of ones has a probability of 1 given the output class. Thus it does not affect the probability of the input data sample given the output class, and in turn, results in the same outputs.

(b) [15 MARKS] Implement logistic regression.

Solution: Implemented as Class **LogisticReg** in the classalgorithms.py file.

The average error obtained is 23.79 with a standard deviation of 0.428 in the errors for 20 runs.

(c) [20 MARKS] Implement a neural network with a single hidden layer, with the sigmoid transfer.

Solution: Implemented as Class **NeuralNet_1hiddenlayer** in the classalgorithms.py file.

The average error obtained is 22.8 with a standard deviation of 0.709 in the errors for 20 runs.

(d) [10 MARKS] Implement k-fold internal cross-validation for picking the best meta-parameters for logistic regression and the neural network. A simple, generic cross-validation interface has been provided in *script_classify.py*. For this assignment, use $k = 5$ folds. Choose at least one meta-parameter and at least three values of that meta-parameter to test for each algorithm. Report the average and standard error for the meta-parameters chosen by cross-validation on the test set.

Solution: Implemented as **cross_validate()** function in the script_classify.py file.

Logistic regression:

I chose the meta-parameter as **stepsize** and chose 4 different values of 0.001, 0.005, 0.01, 0.05.

I got the following average and standard errors for these values by cross-validation (k=5 folds) on the cross validation test sets for 1 run:

Step size = 0.001 had an average error of 26.42 and standard error of 0.698.
Step size = 0.005 had an average error of 23.92 and standard error of 0.656.
Step size = 0.01 had an average error of 23.72 and standard error of 0.682.
Step size = 0.05 had an average error of 24.12 and standard error of 0.481.

For this run, step size of 0.01 had the lowest cross validation average error of 23.72 and it was chosen as the best meta-parameter. Using this as the meta-parameter on the real test data for 20 runs, I got an average error of 23.79 and a standard error of 0.095.

Neural network with 1 hidden layer:

I chose the meta-parameters as **epochs** and **number of nodes (neurons) in the hidden layer** and chose 4 different values mentioned as follows.

I got the following average and standard errors for these values by cross-validation (k=5 folds) on the cross validation test sets for 1 run:

Epochs = 80, Nodes = 4 had an average error of 22.78 and standard error of 0.676.
Epochs = 40, Nodes = 8 had an average error of 22.3 and standard error of 0.445.
Epochs = 20, Nodes = 16 had an average error of 23.32 and standard error of 0.341.
Epochs = 10, Nodes = 32 had an average error of 23.66 and standard error of 0.544.

For this run, epochs = 40, nodes = 8 had the lowest cross validation average error of 22.3 and they were chosen as the best meta-parameters. Using these as the meta-parameters on the real test data for 20 runs, I got an average error of 22.8 and a standard error of 0.158.

Question 2. [40 MARKS]

In this question, you will implement kernel logistic regression. Kernel logistic regression can be derived using the kernel trick, where the optimal solution $\mathbf{w}$ is always a function of the training data $\mathbf{w} = \mathbf{X}^\top \boldsymbol{\alpha}$ for $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{\alpha} \in \mathbb{R}^n$. Therefore, we could instead learn $\boldsymbol{\alpha}$, and whenever we predict on a new value $\mathbf{x}$, the prediction is $\mathbf{x}^\top \mathbf{w} = \mathbf{x}^\top \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{i=1}^n k(\mathbf{x}, \mathbf{x}_i)\alpha_i$ with $k(\mathbf{x}, \mathbf{x}_i) = \langle \mathbf{x}, \mathbf{x}_i \rangle$ in this case. In general, we can extend to other feature representations on $\mathbf{x}$, giving $\phi(\mathbf{x})$ and so a different kernel $k(\mathbf{x}, \mathbf{x}_i) = \langle \mathbf{x}, \mathbf{x}_i \rangle$.

The kernel trick is useful conceptually, and for algorithm derivation. In practice, when implementing kernel regression, we do not need to consider the kernel trick. Rather, the procedure is simple, involving replacing your current features with the kernel features and performing standard regression or classification. For learning, we replace the training data with the new kernel representation:

$$\mathbf{K}_{\text{train}} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{c}_1) & k(\mathbf{x}_1, \mathbf{c}_2) & \dots & k(\mathbf{x}_1, \mathbf{c}_k) \\ \vdots & \vdots & \vdots & \vdots \\ k(\mathbf{x}_n, \mathbf{c}_1) & k(\mathbf{x}_n, \mathbf{c}_2) & \dots & k(\mathbf{x}_n, \mathbf{c}_k) \end{bmatrix} \in \mathbb{R}^{n \times k}$$

for some chosen centers (above those chosen centers were the training data samples $\mathbf{x}_i$). For example, for the linear kernel above with $k(\mathbf{x}, \mathbf{x}_i) = \langle \mathbf{x}, \mathbf{x}_i \rangle$, the center is $\mathbf{c} = \mathbf{x}_i$. Notice that the number of features is now k, the number of selected centers, as opposed to the original dimension d. Once your've transformed your data to this new representation, then you learn $\mathbf{w}$ with logistic regression as usual, such that $\mathbf{K}_{\text{train}}\mathbf{w}$ approximates $\mathbf{y}_{\text{train}}$. As before, you can consider adding regularization. The prediction is similarly simple, where each new point is transformed into a kernel representation using the selected centers.

(a) [25 MARKS] Implement kernel logistic regression with a linear kernel and run it on `susysubset`. Compare the performance in one sentence to the performance of the algorithms from the first question.

Solution: Implemented as Class **LinearKernelLogisticRegression** in the classalgorithms.py file. I have implemented the Linear Kernel as a dot product between a data point and a centre.

When tested on 20 runs, the Linear Kernel Logistic Regression performed worse than the other algorithms (Naive Bayes, Logistic Regression, and Neural Network with 1 hidden layer) as it had higher average error of 29.4 and a much higher standard deviation of 4.19 in the errors than the other algorithms which all had approximately an average error of 24.16 and standard deviation of 0.565 in the errors.

(b) [15 MARKS] Using the same implementation, change the linear kernel to a Hamming distance kernel and run the algorithm on the dataset `Census`. In one or two sentences, summarize your performance, compared to the random predictor.

Solution: Implemented as Class **HammingDistanceKernelLogisticRegression** in the classalgorithms.py file. I have implemented the Hamming distance between a data point and a

centre as the number of different features between them. If they have all same features, then the Hamming distance is zero.

When tested on 20 runs, the Hamming Distance Kernel Logistic Regression had a much lower average error of 24.813 but a very high standard deviation of 9.872 in the errors when compared to the random predictor which had a much higher average error of 50.16 and a much lower standard deviation of 0.702 in the errors. Thus, the Hamming Distance Kernel Logistic Regression seems to perform better than the random predictor on average but its performance is unstable with a very high variance.

Bonus (mandatory for 566). [20 MARKS]

(a) [10 MARKS] In the first part of this assignment, you implemented a neural network with one hidden layer. Implement a second neural network with two hidden layers. Additionally, use your implementation of RMSProp from the previous assignment to train this two hidden layer neural network.

Solution: Implemented as Class **NeuralNet_2hiddenlayers** in the classalgorithms.py file.

I have implemented Adam (and not RMSProp) from the previous assignment. In a discussion on eClass, the TA mentioned implementing either RMSProp or Adam since we did not implement RMSProp in the previous assignment.

The average error obtained is 22.40 with a standard deviation of 0.572 in the errors for 20 runs.

(b) [10 MARKS] In the first part of the assignment, you implement k-fold internal cross-validation. When partitioning the dataset into k validation sets, it is important to balance the ratio of samples belonging to class A versus samples belonging to class B for each validation set.

Implement stratified sampling k-fold cross-validation so each validation set has the same class ratio as the training set. Test each of the algorithms in the first part of the assignment again, and report the differences in your findings.

Solution: Implemented as **stratified_cross_validate()** function in the script_classify.py file.

I did not encounter any significant differences in the validation dataset results for cross validation versus stratified cross validation strategies. It might be because there was not much imbalance in the data distribution over the two classes (about 55:45). I implemented the two strategies for k=5 folds for 20 runs for Naive Bayes, Logistic Regression, and Neural Network with 1 hidden layer for training and test data of 5000 size each. Quite a few times they both chose the same best meta-parameters and only sometimes they chose different ones. The average validation errors on the meta-parameters for both strategies were not significantly different and thus it is not insightful enough to comment anything about which is a better strategy based on my observations on this particular experiment. However, stratified sampling did result in some improvements in terms of variance in the errors.

I think we can have more confidence in the results of stratified cross validation since it maintains

the data with the same proportion of classes in the validation training and test data as that of the training data. As a result of this, the models are trained better and can generalize better on each of the folds, giving us a better and a stable metric of cross validation error. Theoretically, stratified sampling should also have low variance in the errors over multiple folds. This would help in reliably selecting the best meta-parameters. Validation without stratified sampling, may not give a reliable metric over the cross validation error (and in turn, the best meta-parameters) as some of their data folds may be skewed towards one of the classes and hence, may not generalize well on the cross validation test data.

Thus, stratified sampling is more representative of the actual data and hence should be preferred. Its role might be more important when we have larger data size and a significant imbalance in the data distribution over the classes.