Introduction to Data Mining: Project - Basic
Summer 2020

Due: Jun 21st, 23:59:59 CST (UTC +8).

1. A Walk Through Ensemble Models

In this problem, you will implement a whole bunch of ensemble models and compare their performance and properties.

Here we use a *Titanic dataset*. The meaning of each data columns can be found *here*. You are required to predicting the survival of Titanic passengers. For your convenience, your classifiers are only required to handle the discrete variables.

In hw2, we use leave-one-out cross-validation (LOOCV) to choose regularization parameters. Here, we use *k-fold cross-validation*. In k-fold cross-validation, the original sample is randomly partitioned into k equal sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k-1$ subsamples are used as training data.

Skeleton code including *run.ipynb* and Python functions including *tree_plotter.py* are provided for your convenience. Please see the comments in the code for more details. What you need to do is implementing each algorithm and write scripts to play with them. Your algorithm implementations should be able to handle data in arbitrary dimension. See *run.ipynb* for a script example.

(a) **Decision Tree**

Implement decision tree algorithm (in *decision_tree.py*), then answer the following questions.

Implementation hints:

(i) You need to implement three popular criterions for this part, i.e. *information gain*, *information gain ratio*, and *gini impurity*.

(ii) Implementing a stopping criteria is essential in preventing overfitting. Here are several required stopping criteria you need to implement:

i. Limited depth: dont split if the node is beyond the *max_depth* in the tree.

ii. Min data in leaf: dont split if the number of data in a node is smaller than *min_data_leaf*.

(iii) You can randomly choose a subtree if the feature value in testing data not exists in training data.

(iv) You don't need to implement the stuff about *sample_feature* until the **Random Forest** part. We suggest you ignore it first as long as there is no error regarding the *sample_feature*.

(v) You don't need to implement the stuff about *sample_weights* until the **Adaboost** part. We suggest you ignore it first as long as there is no error regarding the *sample_weights*.

Questions:

(i) Train a very shallow decision tree (for example, a depth 2 tree, although you may choose any depth that looks good) and visualize your tree. The visualization method has been implemented and called in your jupyter notebook. Paste the visualization image in your writeup.

(ii) For your above decision tree, and for a data point of your choosing from each class (survived and not survived), state the splits (i.e. which feature and what value of that feature) your decision tree made to classify it. An example of this might look like:
(a) "Title" = 1
(b) "Pclass" = 1
(c) Survived.

(iii) How do the parameters *max_depth* and *min_data_leaf* impact the performance, especially in terms of the underfitting/overfitting trade-off?

(iv) What is the training error rate and validation error rate? Is this underfitting or overfitting?

(v) Choose your best validation error parameters and report the final testing accuracy.

(vi) What knowledge can you gain from the dataset and trained decision tree?

(b) **Random Forest**

Implement Random Forest (in *random_forest.py*), then answer the following questions.

Implementation hints:

(i) You need to implement the *sample_feature* for this part in decision tree. This should be only a few of extra lines of code if implemented appropriately.

Questions

(i) How are your optimal parameters for **decision trees** in **random forest** different from the optimal parameters for a single decision tree? Why?

(ii) What is the training error rate and validation error rate if the number of trees is 10 and 100 respectively?

(iii) How does random forest improve over a single decision tree? Does it improve the bias or variance? Please explain it with your experiment results.

(iv) Choose your best validation error parameters and report the final testing accuracy.

(c) **Adaboost**

Implement Adaboost (in *adaboost.py*), then answer the following questions. Here please use decision tree as the base learner.

Implementation hints:

(i) You need to implement the *sample_weights* for this part in decision tree. *sample_weights* provides an array of weights for each data sample in training data, and this is required for **Adaboost**. To implement this, you need to modify the calculation for criterion (i.e. *entropy*, *information_gain_ratio*, and *gini*), and also the *majority_vote* function. Here we give a brief illustration of how to modify the calculation formula. Take the *entropy* as an example:

Without loss of generality, assume one node is split into two sub-trees, i.e. the left tree(lt), and the right tree(rt). The original entropy after split should be:

$$\frac{|S_{lt}|}{|S|}Entropy(S_{lt}) + \frac{|S_{rt}|}{|S|}Entropy(S_{rt})$$

where the *Entropy* of subtrees are calculated accroding to the formula of *Entropy*. However, with *sample_weights*, the entropy after split is modified as:

$$\frac{W(S_{lt})}{W(S)}Entropy(S_{lt}) + \frac{W(S_{rt})}{W(S)}Entropy(S_{rt})$$

where $W(S)$ means the summation of all the weights of the set S. Also, the calculation of one set S is modified as:

$$Entropy(S) = -\sum_{i}^{c} p_i \log p_i,$$
$$where\ p_i = \sum_{j} I(y_j = i) w_j,$$

where w_j is the *weight* for the j-th sample, y_j is the *label* for the j-th sample. Here, I is an indicator function, which means

$$I(y_j = i) = \begin{cases} 0, & y_j \neq i \\ 1, & y_j = i \end{cases}$$

The modification for *gini impurity* and *information gain ratio* is also similar. To sum up, it can be viewed that the probability of i-th sample is changed from $\frac{1}{n}$ to w_i. You can also have a look at the explanation from stackoverflow here.

(ii) You should vectorize your code, otherwise the calculation may be too slow for this part.

Questions:

(i) People often use **decision stump**, which is essentially decision tree with depth of 1, as the base learner for adaboost. You can also try decision tree with other parameters as the base learner and compare their performance such as error rate, training time, and testing time. Why people prefer decision stump as the base learner for adaboost, while prefer decision tree with larger depth as the base learner for random forest?

(ii) What is the training error rate and validation error rate if the number of trees is 10 and 100 respectively?

(iii) How does Adaboost improve over a single decision tree? Does it improve the bias or variance? Please explain it according to the error rate.

(iv) Choose your best validation error parameters and report the final testing accuracy.

(d) **Comparsion**

Report the result of all learned classifiers like: naive bayes, perception, logistic regression, SVM, neural network, KNN, decision tree, random forest, adaboost (It is better to use your own implementation, or you can use *sklearn* here). And state their advantages and disadvantages.

Please submit your homework report to at http://courses.zju.edu.cn:8060/course/11827/ in pdf format, with all your code in a zip archive.