Machine Learning HW5 – Random Forest

NCTU EE 0310115 鍾文玉, 0310128 游騰杰

Programming language: Python

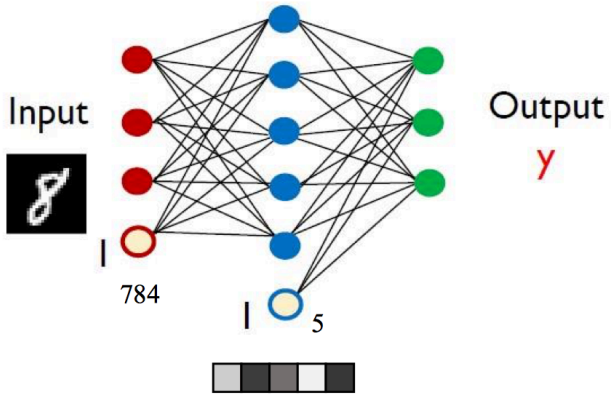
Environment: Python 2.7.13 under Homebrew @ Mac OS X 10.11.6

Python module used: numpy, pandas, scikit-learn, random

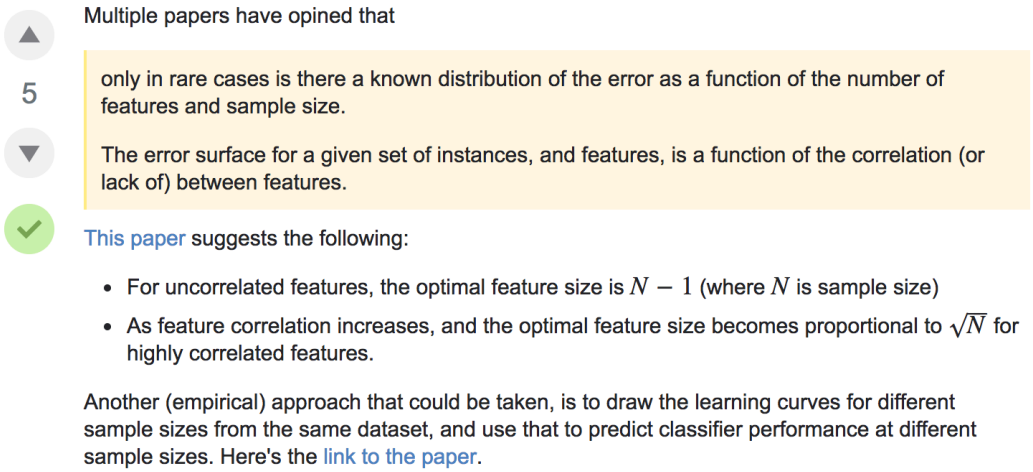
Plot: Matlab R2016b

1. Feature Extractor

Please use a feature extractor to pre-process the input data and send it into the random forest model which is identical to the task 1. A good choice will make your model much more powerful!



As illustrated above, we first train the neural network with one hidden layer, in this experiment, we set the number of nodes in the hidden layer to the square root of the total features (784,) namely, the number of nodes in the hidden layer is set to 28.



Multiple papers have opined that

> only in rare cases is there a known distribution of the error as a function of the number of features and sample size.
>
> The error surface for a given set of instances, and features, is a function of the correlation (or lack of) between features.

This paper suggests the following:

- For uncorrelated features, the optimal feature size is $N - 1$ (where N is sample size)
- As feature correlation increases, and the optimal feature size becomes proportional to $\sqrt{N}$ for highly correlated features.

Another (empirical) approach that could be taken, is to draw the learning curves for different sample sizes from the same dataset, and use that to predict classifier performance at different sample sizes. Here's the link to the paper.

Reference:

Any “rules of thumb” on number of features versus number of instances? (small data sets)
https://datascience.stackexchange.com/questions/11390/any-rules-of-thumb-on-number-of-features-versus-number-of-instances-small-da

After the neural network is trained, we take the weights of all nodes in the hidden layer as feature extractor (one column for each node,) where the first subscript (n) denotes the index of weight in one node, the second subscript (k) denotes the index of the nodes.

$$\begin{bmatrix} w_{1,\ 1} & \cdots & w_{n,\ 1} \\ \vdots & \ddots & \vdots \\ w_{1,\ k} & \cdots & w_{n,\ k} \end{bmatrix}$$

Extracting the feature is simply matrix dot product just like projecting the data to lower dimension subspace (similar to that of PCA does,) the formula is shown below, where each row in matrix Y is the "Extracted" version of the original data.

$$X \cdot W = Y$$

$$\begin{bmatrix} x_{1,\ 1} & \cdots & x_{1,\ n} \\ \vdots & \ddots & \vdots \\ x_{m,\ 1} & \cdots & x_{m,\ n} \end{bmatrix} \cdot \begin{bmatrix} w_{1,\ 1} & \cdots & w_{1,\ k} \\ \vdots & \ddots & \vdots \\ w_{n,\ 1} & \cdots & w_{n,\ k} \end{bmatrix} = \begin{bmatrix} y_{1,\ 1} & \cdots & y_{1,\ k} \\ \vdots & \ddots & \vdots \\ y_{m,\ 1} & \cdots & y_{m,\ k} \end{bmatrix}$$

Implementation:

```
28
29  # Use Neural Network as Feature Extractor
30  neural_network = MLPClassifier(hidden_layer_sizes = (numfeatures, ), \
31  ____________________________________ activation = 'logistic', \
32  ____________________________________ solver = 'adam', \
33  ____________________________________ batch_size = 'auto', \
34  ____________________________________ learning_rate = 'adaptive', \
35  ____________________________________ max_iter = 200, \
36  ____________________________________ shuffle = True, \
37  ____________________________________ verbose = True)
38  neural_network.fit(training_data, training_targ)
39  extractor = neural_network.coefs_[0]
40  train_features = training_data.dot(extractor)
41
```

Here we use Neural Network with one hidden layer and logistic sigmoid function as activation function, the weight of the layer can be accessed by the attribute coefs_[index].

`coefs_` : list, length n_layers - 1

The ith element in the list represents the weight matrix corresponding to layer i.

We leave solver to default 'adam' after reading the documentation.

solver : {'lbfgs', 'sgd', 'adam'}, default 'adam'

The solver for weight optimization.

- 'lbfgs' is an optimizer in the family of quasi-Newton methods.
- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Note: The default solver 'adam' works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.

The Neural Network also uses mini-Batch for the solver with size = min(200, n_samples),

batch_size : int, optional, default 'auto'

Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the classifier will not use minibatch. When set to "auto", batch_size=min(200, n_samples)

which is 200 in our case, also, to optimize the training result, we set "shuffle" to True, this will shuffle samples in each iteration to improve the result.

max_iter : int, optional, default 200

Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations.

Lastly, we set max_iter of the gradient decent to 200, placing the upper bound of iteration on the model to converge, and we set "verbose" to True to observe the training of the Neural Network.

```
Iteration 123, loss = 0.00381861
Iteration 124, loss = 0.00373415
Iteration 125, loss = 0.00362452
Iteration 126, loss = 0.00356968
Iteration 127, loss = 0.00347874
Iteration 128, loss = 0.00340284
Training loss did not improve more than tol=0.000100 for two consecutive
epochs. Stopping.
Number of features extracted: 100
Error rate: 9.52% ( 238/2500)
 - Class 1:   8, Class 2:  13, Class 3: 140, Class 4:  36, Class 5:  41
```

In general, the model stops the iteration at around 100 to 150 depending on the randomly chosen data because the model itself has a threshold (difference of loss between two consecutive epochs) equal to 0.0001 to prevent overfitting to the data.

```
55
56  randomForest = RandomForestClassifier(n_estimators = numTrees, \
57                                        min_samples_leaf = minLeafNode)
58
```

The random forest settings are shown above.

Reference:

sklearn.neural_network.MLPClassifier — scikit-learn 0.18.1 documentation

http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

3.2.4.3.1. sklearn.ensemble.RandomForestClassifier — scikit-learn 0.18.1 documentation

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

2. Random Forest

Please implement a random forest which contains 100 decision trees. There are some settings should be followed:

- The minimum number of samples per leaf node = 1000

- The fraction of samples to be randomly selected with replacement for constructing each decision tree = 50%.

Draw any three decision trees and compute the accuracy of each of them. Show the final accuracy of the random forest model you've trained and explain why the performance doesn't look well?

```
41
42  def subsample_data(data, target, frac):
43      numData = data.shape[0]
44      amount = int(numData * frac)
45
46      sample = list()
47      sample_t = list()
48      for itr in range(amount):
49          index = random.randint(1, numData) - 1
50          sample.append(data[index, :])
51          sample_t.append(target[index])
52
53      sample = np.vstack(sample)
54      sample_t = np.vstack(sample_t)
55      return sample, sample_t.reshape(len(sample_t), )
56
```

We implement the function subsample_data to select data randomly with replacement.

```
54
55  randomForest = RandomForestClassifier(n_estimators = numTrees, \
56                                        min_samples_leaf = minLeafNode)
57
58  randomForest = randomForest.fit(train_features, target)
59  for itr_tree in randomForest.estimators_:
60      filename = "trees/tree_{:d}.dot".format(itr + 1)
61      export_graphviz(itr_tree, out_file=filename)
62
```

To visualize the decision tree we build, we use the function export_graphviz from sklearn.tree to export the decision trees in the random forest.

Randomly chosen three decision trees:

# 19	# 31	# 83
gini = 0.7995 samples = 1593 value = [507, 522, 522, 492, 457]	gini = 0.7995 samples = 1575 value = [484, 520, 527, 509, 460]	gini = 0.7998 samples = 1568 value = [490, 515, 520, 479, 496]

Reference:

sklearn.tree.export_graphviz — scikit-learn 0.18.1 documentation

http://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html#sklearn.tree.export_graphviz

Error Rate = 80%

```
$ python2 HW5_Random_Forest_mod.py
Error rate: 80.00% (2000/2500)
 - Class 1: 500, Class 2: 500, Class 3:   0, Class 4: 500, Class 5: 500
```

Discussion:

The bad result is foreseeable and quite obvious, the minimum number of samples per leaf node (1000 in this case) restrains the classifier to leave at least 1000 samples after the last decision. But the total amount of sample we gave to the classifier is merely 50% of 5000, namely, 2500, so the depth of the tree must be one (two leaf nodes) because the number of samples does not allow a tree to have more than two leaf nodes, which requires at least 2 * 1000 = 2000 samples, not having enough samples, the decision tree does not grow large, in the case of multiclass classification, the model is sure to give very poor regardless of the number of trees.

Further discussion is combined with the parameter max_depth in next section.

3. Parameters Adjustment

Try to find the relationship between the accuracy and the following parameters:

- The number of decision trees.

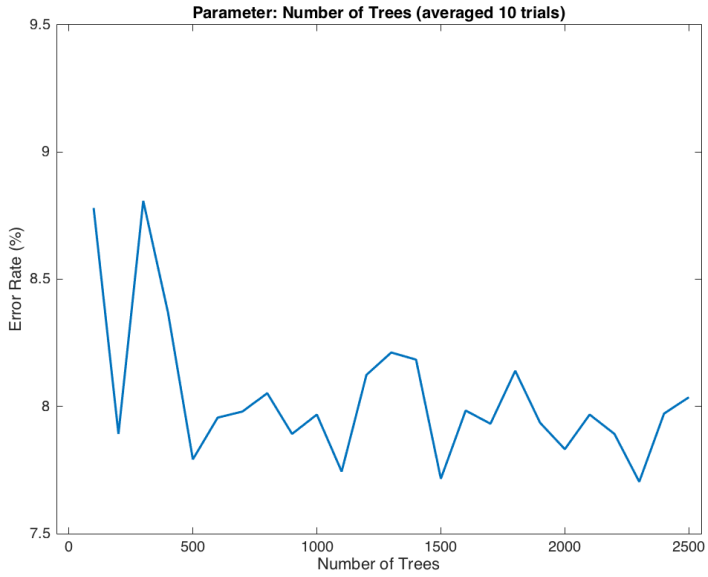
- The minimum number of samples per leaf node.

- The fraction of samples to be randomly selected with replacement for constructing each decision tree.

Plot the accuracy (or error) curve with respect to the parameters.

(1) number of decision trees

(numfeatures = 28, minLeafNode = 1000, sample = 80%)



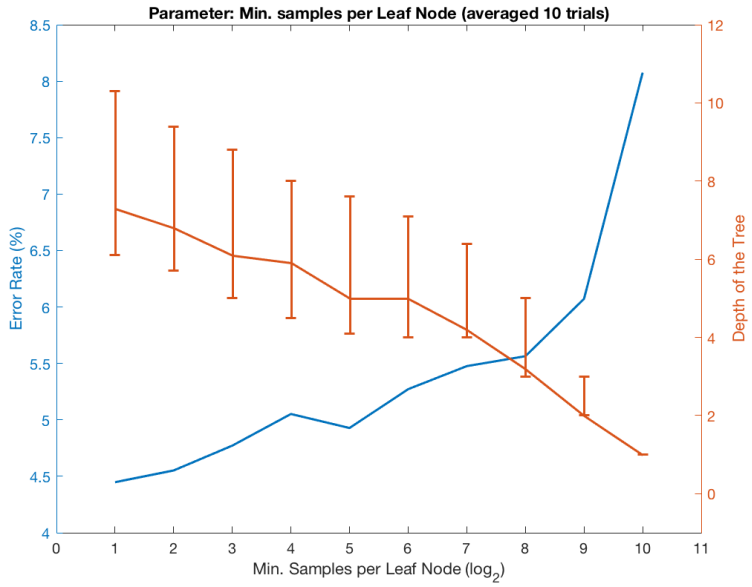
Random Forest is a special case of Bagging, and Bagging is a member of averaging methods in ensemble learning. In averaging methods, we build several estimators independently and then ensemble their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced during averaging.

The idea of Bagging is to build several instances of base estimators (e.g. Decision Trees in Random Forest) on random subsets of the original training set with replacement and then aggregate their individual predictions to reach a final prediction. Variances within base estimators are then reduced by introducing randomization and collective prediction into the model. Bagging methods provide a way to reduce overfitting, they work best with strong and complex models (e.g. fully developed decision trees), in contrast with boosting methods which usually work best with weak models (e.g. shallow decision trees).

Reference:
1.11. Ensemble methods — scikit-learn 0.18.1 documentation
http://scikit-learn.org/stable/modules/ensemble.html

(2) minimum number of samples per leaf node

(numfeatures = 28, numTrees = 100, sample = 80%)



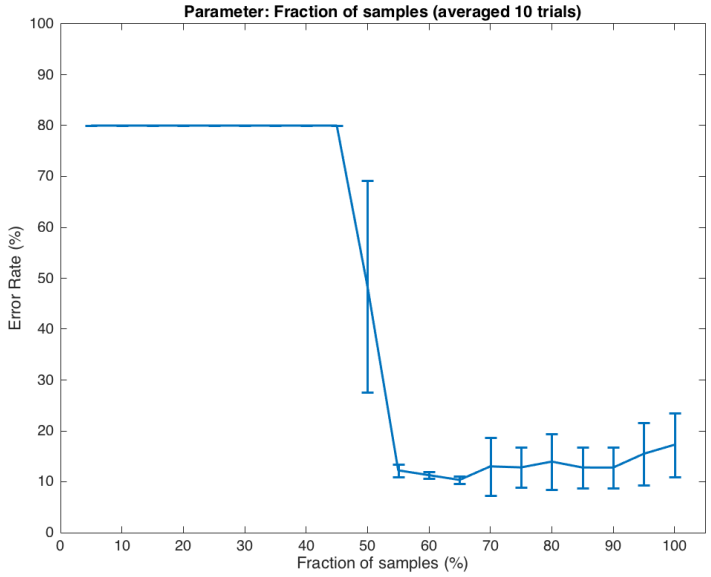
As we can see from the above (upper bound of the error bar correspond to maximum depth and the lower bound correspond to minimum depth in the trials, the curve is the average depth,) in cases where the minimum number of samples per leaf node is low, all three values, average, maximum, and minimum depth are the largest, they descend with the growth of the minimum samples per leaf node.

The idea is quite straightforward, the minimum samples per leaf node indicates how well-fitted the trees are to their individual training dataset, if the tree does not distinguish the data to too detail, the depth of the tree must not be too big and vice versa.

As mentioned in previous page, Bagging enables the strong and complex base estimators to be close enough (overfitting) to the dataset and yields an overall great result. When the number of trees is fixed, the depth and minimum samples per leaf node is in inversely proportional to each other.

(3) fraction of samples

(numfeatures = 28, numTrees = 100, minLeafNode = 1000)



The error bar indicates the standard deviation of the trials we conducted on the model, since Random Forest is a variant of Bagging, we will get different model every time because the randomly selected of training data for each Decision Tree. In the case where the fraction of sample is less than 50% (5000 in total,) the model does not work properly. While we increase the fraction to 50%, the model becomes more possible to classify the testing data with very moderate accuracy (error rate = 48.35%,) the large variance in the trials indicates that the model needs more data to work properly almost every time.

Continuing increasing the fraction of sample yield to better and more stable result in the trials, but not monotonically in all aspect, we conclude that having sufficient and critical data (data with critical features) is crucial to the build a model, but there’ s no formula to use, it requires the knowledge on the case and depends on the application we are dealing with.