George Mason University

Learning From Data   Fall 2024

Computer Exercise #6

Assigned: October 07, 2024
Due Date: October 21, 2024

Overview: This computer exercise is concerned with decision trees, random forests, boosting and bagging. As with previous computer exercises, you should submit your write-up in PDF format. The due date for this is **Monday, October 21** at midnight. No submissions after this time will be accepted.

Comment: This will be your opportunity to solidify your knowledge and understanding of ensemble learning by example. This should be a good preparation for the upcoming midterm exam. Although your report does not need to be submitted until after the exam, reading the assignment and working through some of the exercises will be beneficial.

Computer Exercise 6.1 (Decision Trees):

Decision trees are versatile machine learning algorithms that can used for classification and regression. They are very powerful supervised learning algorithms, capable of fitting complex datasets. However, they must be designed carefully to avoid problems with overfitting. Decision trees are the fundamental component of random forests and boosting algorithms, which are among the most powerful machine learning algorithms in use today.

1. **Getting Started**

   The first step in this exercise is to load a number of classes, all of which you have worked with before,

   ```
   #Common imports
   import numpy as np
   import pandas as pd
   import matplotlib.pyplot as plt
   import seaborn as sns
   ```

   Others for various tasks related to decision trees and boosting will be loaded as they are needed.

2. **The dataset**

   In this computer exercise you will be working with a dataset of eight-bit 28×28 grayscale images of clothing articles, such as T-shirts, dresses, and sneakers. There are ten classes, and the class labels are shown in Table 1.

   Examples of the first nine images in the dataset are shown in Fig. 1.

   (a) The dataset is stored in .csv format. Assuming that the training and testing datasets are in the subfolder `data`, they may be loaded as a pandas `dataframe` as follows:

   ```
   import pandas as pd
   #
   data_train = pd.read_csv('data/train.csv', dtype=int) # read training data
   data_test = pd.read_csv('data/test.csv', dtype=int)   # read test data
   ```

   Questions

   Examine the structure of this dataset using the `head` method,

0	T-Shirt/Top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle Boot

Table 1: The class labels on the clothing dataset.

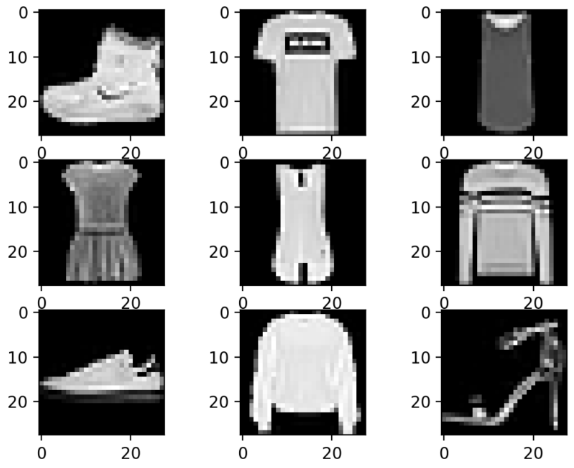

Figure 1: Nine sample images from the dataset. Each is an eight-bit 28×28 image.

```
data_train.head() ; data_test.head()
```

How many images are in the dataset? What are the features and what is the dimension of the feature vectors?

(b) Before it is possible to design a classifier, it is necessary to divide the data into data samples `X` and target values (classes), `y` as follows:

```
X_train = data_train.drop('label', axis=1)
y_train = data_train['label']
X_test = data_test.drop('label', axis=1)
y_test = data_test['label']
```

To get a sense of what the images in the dataset look like, the i^{th} image may be displayed with the commands:

```
i =                        #Put in a value for i
plt.imshow(X_train.iloc[i,:].values.reshape([28,28]))
plt.title('Class %i' %y_train[i])
plt.show()
```

3. **Decision Tree** Now that we have a training and test set along including the images $\boldsymbol{x}$ and the classes y, we may train a decision tree for classification. First will be a deep decision tree, and then we will look at top-down regularization.

(a) Design a deep decision tree classifier using the training data `X_train` and `y_train` created in Part 2. This may be done using the `DecisionTreeClassifier` in the `tree` library of scikit-*learn* and the `fit` method. Begin by using the default values for all hyperparameters as follows:

```
from sklearn.tree import DecisionTreeClassifier
#
Tree_model = DecisionTreeClassifier().fit(X_train,y_train)
```

The accuracy of your model on the training data may be found using the `score` method.

Questions

i. With the default values, what is the structure of the tree? For example, are there any limits on the number of leaves, is the depth of the tree constrained, and so on.
ii. How many leaves are in the tree that is learned, and what is the purity of the leaves? What is the maximum depth of the tree?
iii. What is the training accuracy of the model on the training data? Discuss your results and any conclusions that you may have. Were you surprised by the results?
iv. What is the accuracy of your decision tree on the test set? What does this tell you?

Assignment:

Find the confusion matrix for your classifier on the test set. What does it tell you? Are there any classes that the classifier is have more trouble with than others?

(b) The next step is to estimate the expected error of your classifier using the `cross_val_score` method in the `model_selection` library of scikit-*learn*. Review what cross-validation is and how it is used to estimate the expected error of a classifier. Ths method requires four parameters. The first parameter is `estimator` that defines the classifier that is to be evaluated, which in this case is `Tree_model`. The second and third parameters, `X` and `y`, are the dataset and target values. The last parameter is the number of folds, which is set using the `cv` parameter. As an example, an estimate of the expected error using five-fold cross-validation may be done as follows,

```
from sklearn.model_selection import cross_val_score
accuracies = cross_val_score(estimator=Tree_model, X=X_train, y=y_train, cv=5)
```

The accuracies of the five folds may then be printed along with the average accuracy over all of the folds:

```
print(all_accuracies)
print(all_accuracies.mean())
```

Assignment

Decide how many folds that you would like to use, and use cross-validation to estimate the expected classification error of your decision tree.[1] You probably want to use either five or ten folds. Explain the reason for the number of folds that you use. What is the advantage of choosing 10 folds versus 5? What do your results show? Is there any evidence that your tree is either underfitting or overfitting the data? Explain.

(c) There are many hyperparameters that may be used to limit the growth of the decision tree, including `max_depth`, `min_samples_split` and `min_samples_leaf`. Read the documentation to see what these hyperparameters do, how they work, and what the default values are.

[1]**Note**: Given the size of the dataset, cross-validation may take some time to complete.

Assignment

Design a decision tree with `min_samples_leaf=5` and `max_depth=12`.

i. What is the training error of the classifier?
ii. Estimate the expected error, compare the results to the tree that was learned in part (a).
iii. Find the confusion matrix for your classifier.
    Discuss what you find. Are there any pairs of classes that seem to be confused more often than others? Which ones? Does it make sense?

(d) The settings used in part (3c) were selected arbitrarily, and in training a classifier it is important to be more disciplined in the way that these hyperparameters are selected. One approach would be to randomly set values for these hyperparameters and find the set that results in the best performance. A better approach would be to use a *grid search* as described below.

Grid Search: To perform grid search, import the `GridSearchCV` class, and then create a dictionary of the parameters that are to be searched along with the values that are to be considered. The following is an example of a dictionary with the name `grid_param` that will be used to perform a grid search:

```
grid_param = { 'min_samples_leaf':[1, 2, 3, 4], 'max_depth':[1, 2, 3, 4 ],
               'criterion': ['gini', 'entropy'] }
```

Note that there are three hyperparameters in this dictionary along with the values that are to be evaluated for each parameter (you will need to decide what hyperparameter values to consider, and how many). The Grid Search algorithm tries all possible combinations of parameter values and returns the combination with the highest accuracy. Therefore, in the above example, the algorithm will check 32 combinations ($4 \times 4 \times 2 = 32$).

Once the parameter dictionary is created, the next step is to create an instance of the `GridSearchCV` class. The parameters that must be set are the name of the estimator and the name of the parameter dictionary. Others that may be set are the metric that is to be used to evaluate the classifier and the number of folds in cross validation. An example is given below:

```
gd_sr = GridSearchCV(estimator=Tree_model, param_grid=grid_param,
                     scoring='accuracy', cv=5)
```

Once the `GridSearchCV` class is initialized, the last step is to call the fit method of the class and pass it the training set as shown below,

```
gd_sr.fit(X_train, y_train)
```

Note: This method can take some time to execute because there are 32 combinations of parameters and a 5-fold cross validation. Therefore there will be a total of 160 trees that are learned.

Once the grid search is done, the next step is to get the parameters that give the highest accuracy. This may be done by printing the `best_params_ attribute` of the `GridSearchCV` object,

```
best_parameters = gd_sr.best_params_
print(best_parameters)
```

The last and final step is to find the accuracy of the tree using the parameters found in the grid search.

```
best_result = gd_sr.best_score_
print(best_result)
```

You may also print all scores from the grid search,

```
allscores=gd_sr.cv_results_['mean_test_score']
print(allscores)
```

Assignment:

Use a grid search to find the best parameters for a decision tree classifier on the dataset `X_train, y_train`. Note: the hyperparameters and the numbers given in the example above were for illustration purposes only, Determine **which parameters** to perform a grid search on along with **the values** to be evaluated. Report the best set of parameter values and the accuracy of the classifier. **Note:** If your grid search is taking too long, you may consider reducing the number of hyperparameters that are being evaluated.

Computer Exercise 6.2 (Random Forests):

The primary drawback of decision trees is that they will overfit the training data without some form of regularization, such as limiting the number of leaves or the minimum number of instances required before a node is split. A random forest is another way to address the overfitting problem. A random forest is an ensemble of decision trees, where each tree is slightly different from the others, and the predictions made by each tree are aggregated to form a final prediction. With many trees in an ensemble, even though each one individually may overfit the data, each one overfits in a different way. By averaging the predictions, overfitting is reduced along with a reduction in the variance.

1. **Random Forest** Random forests get their name from the randomness that is inserted into the growth of each tree, and this randomness ensures that each tree is different. There are two ways in which the trees are randomized. The first is by selecting the data points that are used to build the tree (bootstrap sampling) and the second is by randomly selecting the features that are to be considered in the splitting of a node of the tree.

   A random forrest classifier may be designed using the `RandomForestClassifier` class. An example of how this class is used is given below,

   ```
   from sklearn.ensemble import RandomForestClassifier
   forest_clf = RandomForestClassifier(n_estimators=100, max_depth=2, random_state=0)
   forest_clf.fit(X_train, y_train)
   ```

   There are a number of important hyperparameters that can be specified in the random forest classifier. The first is `n_estimators` (default value is 100), which is the number of trees the algorithm builds. In general, a higher number of trees increases the performance of the random forest and makes predictions more stable, but it also slows down the computation. There is no general rule that specifies the optimum number of trees, and the best number depends on the data.

   Another important parameter is `max_features`, which is the number of features to consider when looking for the best split of a node. Scikit-*Learn* provides several options that are described in the documentation, which includes the default value of the square root of the number of features. There are a number of other hyperparameters that may be defined, such as `min_samples_leaf`, which is the minimum number of samples required to be a leaf node, `max_depth`, the maximum depth of the tree, and `min_samples_split`, the minimum number of samples required to split an internal node.

   Note: The default values for the parameters controlling the size of the trees (e.g. `max_depth`, `min_samples_leaf`, etc.) lead to fully grown and unpruned trees, which can potentially be very large on some datasets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting these parameter values.

Since each tree in the random forest is constructed from boostrap samples, approximately 30% of the training set is not used in the design of each tree. Since a predictor never sees these **out-of-bag** (OOB) samples, the predictor can be evaluated on these instances, without the need for a separate validation set or cross-validation. In Scikit-Learn, the random forest may be evaluated by averaging the performance on the out-of-bag samples for each tree. The score obtained using the out-of-bag samples is obtained in the `oob_score_` attribute. To get this score, it is necessary to set `oob_score_=True`.

Assignment:

(a) Design a random forest classifier for the dataset of clothing articles. Decide what the best values are to use for the hyperparameters (do not necessarily accept the default values). Use the out-of-bag samples to estimate the expected error in your random forest. How does this error compare to the training error?

(b) Investigate the effect of the number of trees and the number of features used in the design of each tree on the performance of your classifier. Describe/document what you find.

(c) Compare your classifier to others that you have designed in terms of performance as well as computational complexity when performing classifications.

2. **Extremely Randomized Trees (Extra Trees)**

With extremely randomized trees, randomness of the trees is achieved in the way splits are computed. As in random forests, a random subset of candidate features is selected, but instead of looking for the most discriminative thresholds, the thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule. More specifically, for each randomly sampled feature, the minimum and maximum values of this feature for all samples in the training set are found, and a number θ, which serves as the decision threshold, is randomly selected between these two extreme values. The data samples are then separated into two classes with respect to θ. An example call to the extra trees class in scikit-*learn* is given below. Generally, the data used to train the Extra Trees do not come from bootstrap samples, but work with the entire dataset.

```
from sklearn.ensemble import ExtraTreesClassifier
#
et_clf = ExtraTreesClassifier(n_estimators=10, max_depth=None, min_samples_split=2)
et_clf.fit(X_Train, y_train)
```

Assignment:
Read the documentation on `ExtraTreesClassifier` to see what hyperparameters that you may define, and repeat parts (a) and (b) in the random forest exercise above. Note that if you want to use oob samples to estimate the expected error, it is necessary to set `oob_score=True` because the default is to use the whole dataset to build the tree. You will also want to determine the appropriate number of estimators (trees) to use.

Computer Exercise 6.3 (Boosting):

Boosting refers to an ensemble method that combines weak learners to form a strong learner. The general idea is to train predictors sequentially, with each one correcting the one before. There are many boosting methods, but one of the most popular is AdaBoost (Adaptive Boosting).

1. **AdaBoost**

With AdaBoost, a new predictor is designed so that it pays more attention to the training instances that its predecessor incorrectly classified. This results in new predictors that focus more on the hard cases.

To build an AdaBoost classifier, a base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The relative weights of misclassified training instances are then increased, and those that are correctly classified are decreased. A second base classifier is then trained using the updated weights and again makes predictions on the training set, the weights are updated, and the processes is repeated until the maximum number of base classifiers is reached.

The only requirement on the weak learner, or base classifier, is that it performs slightly better than a random guess, i.e., it results in an error rate less than 0.5. The choice of a base classifier is not always obvious, and several base classifiers have been proposed. A popular choice is the *decision stump*, which is a single-node classification tree. It should be clear that a decision stump will always have a classification error of less than 50%.

A classifier using the AdaBoost algorithm may be designed using the `AdaBoostClassifier` class in scikit-*learn*. The two key parameters are the number of estimators to use, specified by the values set in `n_estimators`, and the base classifier, which is defined by `base_estimator`. The default base estimator is a decision stump.

An example of how to train a classifier with the AdaBoost algorithm using a maximum of 200 decision stumps is given below:

```
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(n_estimators=200)
ada_clf.fit(X_train, y_train)
```

Read the documentation for `AdaBoostClassifier` to see what other parameters there are and to see what attributes and methods are available for this class.

Assignment

(a) Use `AdaBoostClassifier` to design a strong classifier using the clothing dataset created in part (2b). Use a decision stump as the base classifier, and set `n_estimators=3000`.

(b) Determine the accuracy of your model on the training data and estimate the expected error using cross-validation.

(c) Analyze and comment on your results. How do your results change if you were to use a decision tree with a maximum depth of two or four instead of one as you have with a decision stump classifier? Discuss your findings.

(d) How much is your design affected by the number of estimators?

2. Gradient Boosting

Gradient boosting is another powerful boosting method. The main idea behind gradient boosting is again to additively combine many simple models (in this context known as weak learners), like shallow trees. Gradient boosted trees are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameters are set correctly. Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the `learning_rate`, which controls how strongly each tree tries to correct the mistakes of the previous trees. A higher learning rate means each tree can make larger corrections. Adding more trees to the ensemble, which can be accomplished by increasing `n_estimators`, also increases the model complexity, as the model has more chances to correct mistakes on the training set. An example of how to use `GradientBoostingClassifier` on a dataset that is stored in `X_train` and `y_train` is

```
from sklearn.ensemble import GradientBoostingClassifier
#
gb_clf = GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gb_clf.score(X_train, y_train)))
```

By default, 100 trees with a maximum depth of 3 and a learning rate of 0.1 are used.

Assignment:
Use gradient boosting to design a classifier using your clothing dataset. Describe/document what you find, and comment on how well your classifier peforms compared to the other classifiers that you have designed.

Computer Exercise 6.4 (Final Evaluation):

In the previous exercises, you have designed classifiers using Decision Trees, AdaBoost, Random Forests, Extra Trees and Gradient Boosting.

Assignment:
Based on what you determine to be the best classifier and the best set of hyperparameters for that classifier, evaluate the error on the test set, `X_test, y_test` that you have not yet used in any decisions or design.