Internet Advertisements Prediction

Using Machine Learning Algorithms

Ehab Mohamed
Lewis University
Illinois, United States

***Abstract*—The project aims to recognize the advertisement images shown in web pages. Two supervised machine learning algorithms are used to predict whether the image shown in a web page is an advertisement or not. The two algorithms are Random Forests and Gradient Boosting Classifier. A set of possible advertisements on internet pages is used for training and testing the machine learning models. The performance results of both algorithms show a high prediction accuracy of 97%.**

Keywords—internet advertisements; prediction; machine learning; random forests; gradient boosting classifier; bagging; boosting; ensemble.

I. Objectives

The project aims to predict whether an image shown in a web page is an advertisement or not. The project, which is implemented using python, utilizes two different supervised machine learning algorithms in order to perform the prediction task. Such prediction may aid in learning how to detect and remove internet advertisements appearing in web pages.

II. Dataset Description

The used dataset represents a set of possible advertisements on internet pages and is created by Nicholas Kushmerick in 1998. The dataset is hosted and can be found on the Machine Learning Repository for University of California, Irvine at:

http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements

The dataset consists of 3,279 instances and 1,558 features (i.e., attributes). The features encode the geometry of the image (if available) along with the phrases occurring in the URL, image's URL and alt text, the anchor text, and words occurring near the anchor text.

Each instance in the dataset is classified either as an advertisement (i.e., "ad") or not (i.e., "nonad"). The number of instances classified as nonad is 2,821 (i.e., 86%), whereas the number of instances classified as ads is 458 (i.e., 14%). Furthermore, there is 28% of the instances have missing values in some of the continuous attributes (interpreted as "unknown" or "?").

A. Attributes

The dataset has only 3 continuous attributes and the others are binary. Table I lists the details of the different attributes / features which present in the dataset.

TABLE I. DATA ATTRIBUTES

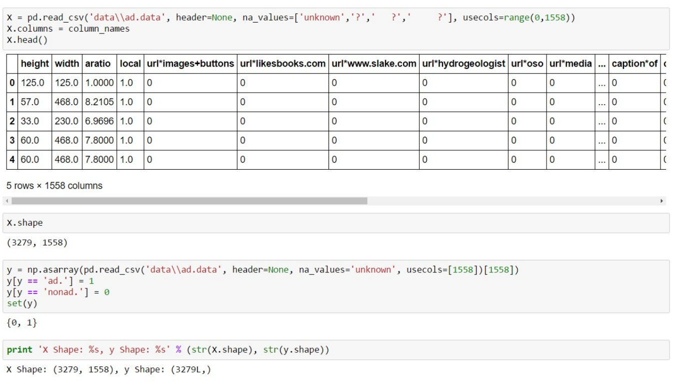
Attribute	*Type*	*Has Missing Values*	*Attribute Name Example*
height	Continuous	Yes	height
width	Continuous	Yes	width
aratio	Continuous	Yes	aratio
local	Binary	No	local
457 features from url terms, each of the form "url*term1+term2..."	Binary	No	url*images+buttons
495 features from origurl terms, in same form	Binary	No	origurl*labyrinth
472 features from ancurl terms, in same form	Binary	No	ancurl*search+direct
111 features from alt terms, in same form	Binary	No	alt*your
19 features from caption terms	Binary	No	caption*and

B. Pre-Processing

The first stage in the project is the pre-processing stage which aims to prepare the data in a way that can be smoothly processed by the machine learning algorithms. This stage consists of five different tasks, as explained next, which are reading data, splitting data into training and testing datasets, handling missing values, and selecting features.

1) Reading Data: aims to read data from the csv source file. In this task, the missing values are interpreted and considered while reading data. Moreover, the classification column is transformed to a binary format (i.e., "ad"=1, "nonad"=0). Fig. 1 illustrates the code of the reading task as well as a sample of the fetched data.

Fig. 1. Reading data and sample preview.



```
X = pd.read_csv('data\\ad.data', header=None, na_values=['unknown','?','   ?','     ?'], usecols=range(0,1558))
X.columns = column_names
X.head()
```

	height	width	aratio	local	url*images+buttons	url*likesbooks.com	url*www.slake.com	url*hydrogeologist	url*oso	url*media	...	caption*of
0	125.0	125.0	1.0000	1.0	0	0	0	0	0	0	...	0
1	57.0	468.0	8.2105	1.0	0	0	0	0	0	0	...	0
2	33.0	230.0	6.9696	1.0	0	0	0	0	0	0	...	0
3	60.0	468.0	7.8000	1.0	0	0	0	0	0	0	...	0
4	60.0	468.0	7.8000	1.0	0	0	0	0	0	0	...	0

5 rows × 1558 columns

```
X.shape
```

```
(3279, 1558)
```

```
y = np.asarray(pd.read_csv('data\\ad.data', header=None, na_values='unknown', usecols=[1558])[1558])
y[y == 'ad.'] = 1
y[y == 'nonad.'] = 0
set(y)
```

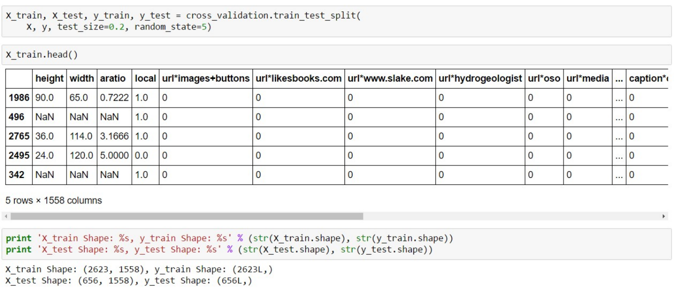
```
{0, 1}
```

```
print 'X Shape: %s, y Shape: %s' % (str(X.shape), str(y.shape))
```

```
X Shape: (3279, 1558), y Shape: (3279L,)
```

2) Splitting Data into Training and Testing Datasets: aims to generate two different datasets from the original dataset, one for training and the other for testing. The training dataset is 80% (i.e., 2,623 instances) of the original dataset, while the testing dataset is 20% (i.e., 656 instances). Fig. 2 illustrates the code of the splitting task as well as a sample of the transformed training dataset.

Fig. 2. Splitting data into training and testing datasets.



```
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
    X, y, test_size=0.2, random_state=5)
```

```
X_train.head()
```

	height	width	aratio	local	url*images+buttons	url*likesbooks.com	url*www.slake.com	url*hydrogeologist	url*oso	url*media	...	caption*
1986	90.0	65.0	0.7222	1.0	0	0	0	0	0	0	...	0
496	NaN	NaN	NaN	1.0	0	0	0	0	0	0	...	0
2765	36.0	114.0	3.1666	1.0	0	0	0	0	0	0	...	0
2495	24.0	120.0	5.0000	0.0	0	0	0	0	0	0	...	0
342	NaN	NaN	NaN	1.0	0	0	0	0	0	0	...	0

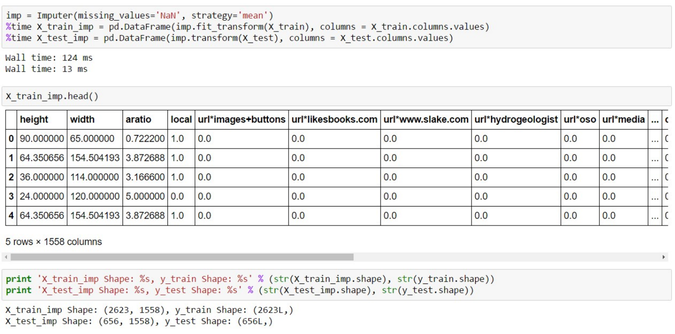
5 rows × 1558 columns

```
print 'X_train Shape: %s, y_train Shape: %s' % (str(X_train.shape), str(y_train.shape))
print 'X_test Shape: %s, y_test Shape: %s' % (str(X_test.shape), str(y_test.shape))
```

```
X_train Shape: (2623, 1558), y_train Shape: (2623L,)
X_test Shape: (656, 1558), y_test Shape: (656L,)
```

3) Handling Missing Values: aims to handle the missing values in the three continuous attributes by replacing the missing values in the training dataset with the mean of the corresponding attribute column. Then the task transforms the testing dataset using the same means generated from the training dataset. Fig. 3 illustrates the code of the missing values handling task as well as a sample of the transformed training dataset.

Fig. 3. Handling the missing values.



```
imp = Imputer(missing_values='NaN', strategy='mean')
%time X_train_imp = pd.DataFrame(imp.fit_transform(X_train), columns = X_train.columns.values)
%time X_test_imp = pd.DataFrame(imp.transform(X_test), columns = X_test.columns.values)
```

```
Wall time: 124 ms
Wall time: 13 ms
```

```
X_train_imp.head()
```

	height	width	aratio	local	url*images+buttons	url*likesbooks.com	url*www.slake.com	url*hydrogeologist	url*oso	url*media	...
0	90.000000	65.000000	0.722200	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...
1	64.350656	154.504193	3.872688	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...
2	36.000000	114.000000	3.166600	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...
3	24.000000	120.000000	5.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...
4	64.350656	154.504193	3.872688	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...

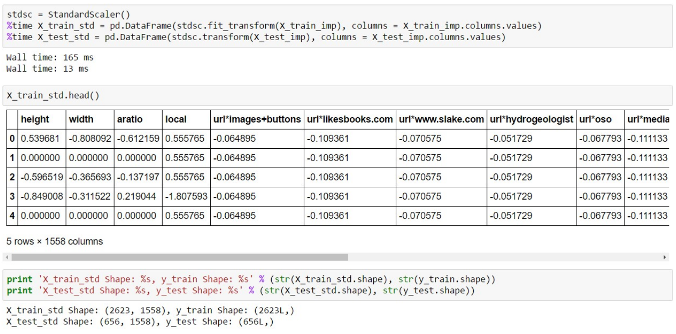
5 rows × 1558 columns

```
print 'X_train_imp Shape: %s, y_train Shape: %s' % (str(X_train_imp.shape), str(y_train.shape))
print 'X_test_imp Shape: %s, y_test Shape: %s' % (str(X_test_imp.shape), str(y_test.shape))
```

```
X_train_imp Shape: (2623, 1558), y_train Shape: (2623L,)
X_test_imp Shape: (656, 1558), y_test Shape: (656L,)
```

4) Standardizing Data: aims to standardize the data by subtracting the mean of the corresponding attribute from each data value and dividing it by the standard deviation. Fig. 4 illustrates the code of the standardizing task as well as a sample of the transformed training dataset.

Fig. 4. Standardizing data.



```
stdsc = StandardScaler()
%time X_train_std = pd.DataFrame(stdsc.fit_transform(X_train_imp), columns = X_train_imp.columns.values)
%time X_test_std = pd.DataFrame(stdsc.transform(X_test_imp), columns = X_test_imp.columns.values)
```

```
Wall time: 165 ms
Wall time: 13 ms
```

```
X_train_std.head()
```

	height	width	aratio	local	url*images+buttons	url*likesbooks.com	url*www.slake.com	url*hydrogeologist	url*oso	url*media
0	0.539681	-0.808092	-0.612159	0.555765	-0.064895	-0.109361	-0.070575	-0.051729	-0.067793	-0.111133
1	0.000000	0.000000	0.000000	0.555765	-0.064895	-0.109361	-0.070575	-0.051729	-0.067793	-0.111133
2	-0.596519	-0.365693	-0.137197	0.555765	-0.064895	-0.109361	-0.070575	-0.051729	-0.067793	-0.111133
3	-0.849008	-0.311522	0.219044	-1.807593	-0.064895	-0.109361	-0.070575	-0.051729	-0.067793	-0.111133
4	0.000000	0.000000	0.000000	0.555765	-0.064895	-0.109361	-0.070575	-0.051729	-0.067793	-0.111133

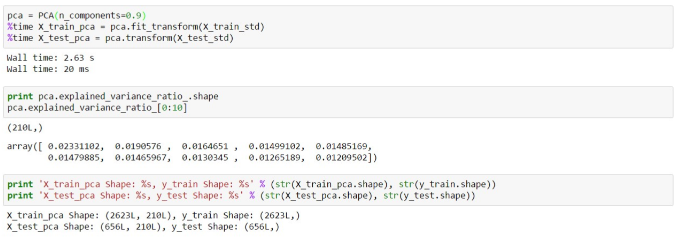
5 rows × 1558 columns

```
print 'X_train_std Shape: %s, y_train Shape: %s' % (str(X_train_std.shape), str(y_train.shape))
print 'X_test_std Shape: %s, y_test Shape: %s' % (str(X_test_std.shape), str(y_test.shape))
```

```
X_train_std Shape: (2623, 1558), y_train Shape: (2623L,)
X_test_std Shape: (656, 1558), y_test Shape: (656L,)
```

5) Selecting Features Using Principle Component Analysis (PCA): aims to reduce the number of features by selecting the features which explain 90% of the variance (i.e., 210 features). The task utilizes the PCA in order to identify the required features and to transform the dataset accordingly. After performing this task, the number of features has been reduced from 1,558 to only 210 features. Fig. 5 illustrates the code of the features selection task.

Fig. 5. Selecting features using PCA.



```
pca = PCA(n_components=0.9)
%time X_train_pca = pca.fit_transform(X_train_std)
%time X_test_pca = pca.transform(X_test_std)
```

```
Wall time: 2.63 s
Wall time: 20 ms
```

```
print pca.explained_variance_ratio_.shape
pca.explained_variance_ratio_[0:10]
```

```
(210L,)

array([ 0.02331102,  0.0190576 ,  0.0164651 ,  0.01499102,  0.01485169,
        0.01479885,  0.01465967,  0.0130345 ,  0.01265189,  0.01209502])
```

```
print 'X_train_pca Shape: %s, y_train Shape: %s' % (str(X_train_pca.shape), str(y_train.shape))
print 'X_test_pca Shape: %s, y_test Shape: %s' % (str(X_test_pca.shape), str(y_test.shape))
```

```
X_train_pca Shape: (2623L, 210L), y_train Shape: (2623L,)
X_test_pca Shape: (656L, 210L), y_test Shape: (656L,)
```

III. Data Mining Process

In the data mining process, two supervised machine learning algorithms are used to perform the prediction task. The first algorithm is the Random Forests (RF), while the other algorithm is the Gradient Boosting Classifier (GBC). These two algorithms are chosen because each one belongs to a different type of ensemble methods.

The main reason behind using ensembles of learners is that there is no single learning algorithm that induces the most accurate learner. Thus, generating a group of base-learners which can be combined to produce higher accuracy.

RF is considered one of the popular applications of the bagging ensemble method. Whereas, GBC is one the popular boosting ensemble methods.

Bagging is considered a voting method where base-learners are trained over slightly different training sets which are generated from the given sample using bootstrapping. For a given training set X of size N, N instances are drawn randomly from X with replacement.

In case of RF, it fits many trees against different samples of the data and averages them together.

Boosting refers to actively generating complementary base-learners by training the next learner after solving the errors of the prior learners.

In case of GBC, it fits successive trees where each solves for the net error of the prior trees.

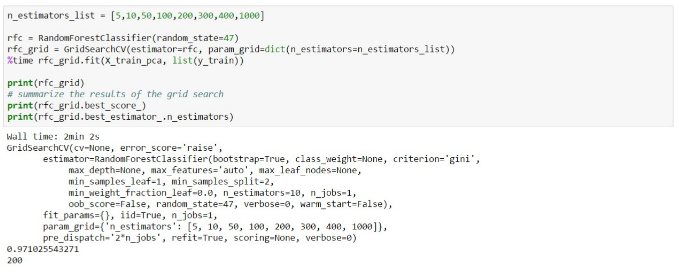
The execution process of both algorithms utilizes a cross-validation method (i.e., GridSearchCV) which implements a "fit" and a "score" task for the decision trees.

A. *Random Forests (RF) Algorithm*

In RF, the number of trees in the forest is provided to the cross-validation method as a list of estimators (5, 10, 50, 100, 200, 300, 400, 1000). The bootstrap option is enabled by default. The execution results show that the best number of trees is 200.

Fig. 6 shows the execution results of the RF algorithm.

Fig. 6. Executing the RF algorithm.



```
n_estimators_list = [5,10,50,100,200,300,400,1000]

rfc = RandomForestClassifier(random_state=47)
rfc_grid = GridSearchCV(estimator=rfc, param_grid=dict(n_estimators=n_estimators_list))
%time rfc_grid.fit(X_train_pca, list(y_train))

print(rfc_grid)
# summarize the results of the grid search
print(rfc_grid.best_score_)
print(rfc_grid.best_estimator_.n_estimators)
```

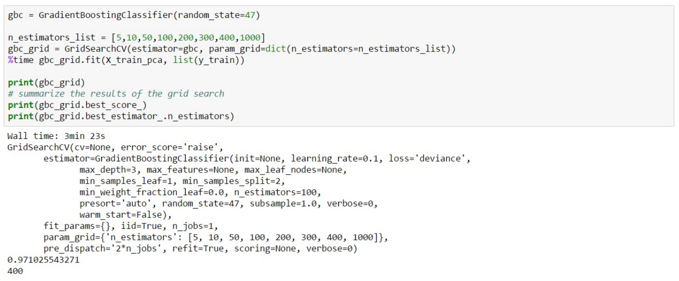
```
Wall time: 2min 2s
GridSearchCV(cv=None, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
            oob_score=False, random_state=47, verbose=0, warm_start=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'n_estimators': [5, 10, 50, 100, 200, 300, 400, 1000]},
       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
0.971025543271
200
```

B. *Gradient Boosting Classifier (GBC) Algorithm*

In GBC, the same number of trees, as RF, is provided to the cross-validation method as a list of estimators (5, 10, 50, 100, 200, 300, 400, 1000). The maximum depth option is kept as default (i.e., 3). The execution results show that the best number of trees is 400.

Fig. 7 shows the execution results of the GBC algorithm.

Fig. 7. Executing the GBC algorithm.



```
gbc = GradientBoostingClassifier(random_state=47)

n_estimators_list = [5,10,50,100,200,300,400,1000]
gbc_grid = GridSearchCV(estimator=gbc, param_grid=dict(n_estimators=n_estimators_list))
%time gbc_grid.fit(X_train_pca, list(y_train))

print(gbc_grid)
# summarize the results of the grid search
print(gbc_grid.best_score_)
print(gbc_grid.best_estimator_.n_estimators)
```

```
Wall time: 3min 23s
GridSearchCV(cv=None, error_score='raise',
       estimator=GradientBoostingClassifier(init=None, learning_rate=0.1, loss='deviance',
              max_depth=3, max_features=None, max_leaf_nodes=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=47, subsample=1.0, verbose=0,
              warm_start=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'n_estimators': [5, 10, 50, 100, 200, 300, 400, 1000]},
       pre_dispatch='2*n_jobs', refit=True, scoring=None, verbose=0)
0.971025543271
400
```

IV. Results

Both RF and GBC algorithms achieved more than 97% of accuracy. However, RF algorithm performed slightly better than the GBC algorithm.

A. *RF Performance*

The performance result of the RF algorithm shows TN = 562, TP = 78, FN = 12 and FP = 4. This result produces a precision measure for the positive classification (i.e., "ad"), which refers to the ratio between the number of TP and the number of total positive predictions (i.e., TP + FP), of 95%.

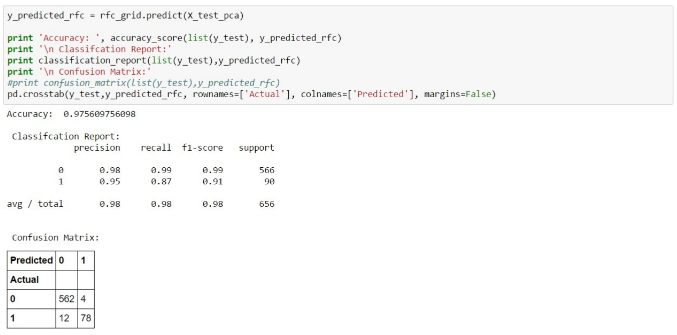
The result shows a recall measure for the positive classification (i.e., "ad"), which refers to the ratio between the number of TP and the number of total actual positives (i.e., TP + FN), of 87%.

Moreover, the result shows that the F1-Score, which combines both precision and recall measures together, reached 91%.

Finally, the result shows that the accuracy of the RF algorithm almost reached 97.6%.

Fig. 8 illustrates the performance report of the executed RF algorithm.

Fig. 8. The RF performance report.



```
y_predicted_rfc = rfc_grid.predict(X_test_pca)

print 'Accuracy: ', accuracy_score(list(y_test), y_predicted_rfc)
print '\n Classifcation Report:'
print classification_report(list(y_test),y_predicted_rfc)
print '\n Confusion Matrix:'
#print confusion_matrix(list(y_test),y_predicted_rfc)
pd.crosstab(y_test,y_predicted_rfc, rownames=['Actual'], colnames=['Predicted'], margins=False)
```

```
Accuracy:  0.975609756098

 Classifcation Report:
             precision    recall  f1-score   support

          0       0.98      0.99      0.99       566
          1       0.95      0.87      0.91        90

avg / total       0.98      0.98      0.98       656


 Confusion Matrix:
```

Predicted	0	1
Actual		
0	562	4
1	12	78

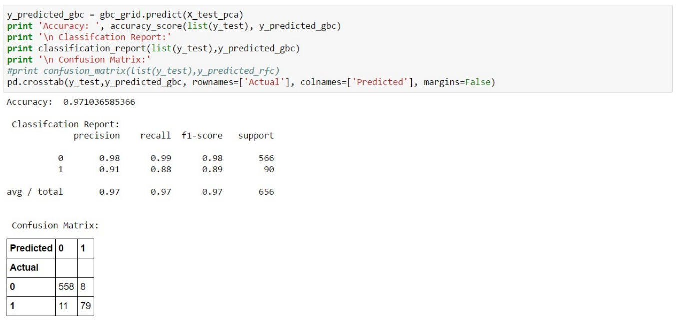
B. *GBC Performance*

The performance result of the GBC algorithm shows TN = 558, TP = 79, FN = 11 and FP = 8. This result produces a precision measure for the positive classification (i.e., "ad") of 91%.

The result shows a recall measure for the positive classification (i.e., "ad") of 88%. Moreover, the result shows that the F1-Score reached 89%.

Finally, the result shows that the accuracy of the GBC algorithm almost reached 97.1%. Fig. 9 illustrates the performance report of the executed GBC algorithm.

Fig. 9. The GBC performance report.



```
y_predicted_gbc = gbc_grid.predict(X_test_pca)
print 'Accuracy: ', accuracy_score(list(y_test), y_predicted_gbc)
print '\n Classifcation Report:'
print classification_report(list(y_test),y_predicted_gbc)
print '\n Confusion Matrix:'
#print confusion_matrix(list(y_test),y_predicted_rfc)
pd.crosstab(y_test,y_predicted_gbc, rownames=['Actual'], colnames=['Predicted'], margins=False)
```

```
Accuracy:  0.971036585366

 Classifcation Report:
             precision    recall  f1-score   support

          0       0.98      0.99      0.98       566
          1       0.91      0.88      0.89        90

avg / total       0.97      0.97      0.97       656


 Confusion Matrix:
```

Predicted	0	1
Actual		
0	558	8
1	11	79

V. Conclusion

The results show that both RF and GBC algorithms reached a high percentage of accuracy which is greater than 97%. Table II summarizes the performance of RF algorithm versus the GBC algorithm.

TABLE II. RF vs GBC Performance

Performance Measure	*RF*	*GBC*	*RF vs GBC*
Precision	95%	91%	**4%** ↑
Recall	87%	88%	**1%** ↓
F1-Score	91%	89%	**2%** ↑
Accuracy	97.6%	97.1%	**0.5%** ↑

This project helped in developing a complete cycle of prediction process. Starting from data pre-processing and reaching building the final prediction model, the process aids a lot in understanding how ensemble methods like Random Forests (i.e., a bagging method) and Gradient Boosting Classifier (i.e., a boosting method) can achieve high prediction performance.