Introduction To Machine Learning Part 2 - Classification

Introduction

The focus of this article is to introduce another class of machine learning algorithms. This class of algorithms is used to build models whose purpose is to predict a discrete valued target random variable. The set of discrete values can be binary or a set of category values. In the case where those values are non-numeric labels (different categories e.g. colors = {blue, red, green} or days = {Monday to Friday}), they can be converted to numbers and used in these algorithms. Unlike the continuous (fractional integer) target values being predicted in regression models, these algorithms attempt to predict whole numbers and are referred to as classification models.

This article [1] described a series of 10 steps that the general process of machine learning follows. The previous article [2] focused on steps 7-10 and how they are fulfilled by regression models. This article will map those steps into classification model construction.

Step 7 is classifier algorithm selection for model training (e.g. KNN, decision trees, random forests and naive bayes). Step 8 is to apply the model to predict a value. Evaluation is done in Step 9 with a confusion matrix and classification report. Finally, Step 10 is optimization. This is done by exploring a new type of "boosting" algorithms (e.g. AdaBoost and XGBoost), instead of random forests, which are a type of "bagging" algorithm.

Lastly is the introduction of a new dataset to analyze. This dataset is unusal and has different properties and statistics that are used in the experiments. This dataset [5] describes pulsars [3], how to identify them [4], how the data was collected and the specific features captured. Finally, the Boston housing dataset is revisited and is analyzed in the appendix.

As with previous articles, this resource [11] was summarized in the descriptions of the sections that follow. All the source code that was reproduced is provided [12] so readers can practice their understanding and self-demonstrate the functionality to gain insights with this educational learning exercise.

Theory

In the previous regression article, the idea was to predict a continuous target variable. The purpose of classification models is to predict a discrete target variable. These algorithms (known as classifiers) opearate on the set of features in the training matrix and attempt to decide on a single output that is a member of a finite set of values. In regression, the approach is to fit a line to a set of data points and minimize the error in terms of distance from that line in regression. For classification, the approach is to process the features and group them into a set of categories. The operations on the matrix and column vector might produce a real (decimal) number, but the final function that processes the output value, results in converting that real value into one value from a finite set of values that represent the target being predicted. As described in the introduction, classification is the problem being addressed by those machine learning algorithms implemented to output a single discrete value instead of a continuous value.

The main algorithm used first in classification is known as Logistic Regression [7]. The purpose of the algorithm is to bin values into one class or another. This is done using the sigmoid equation [8]. The plotted sigmoid curve crosses the vertical y-axis at some intercept value. The target values produced by the model that are being classified are going to have real values above or below that intercept. This means the probability a value is 1 or 0 is a function of where it intercepts the sigmoid curve. This turns a real value into a discrete predicted value.

A very different kind of algorithm is Naive Bayes. It is based on Bayes' theorem involving conditional probabilities of independent and identically distributed events [9]. The distribution of a random variable (or in this case columns in the feature matrix) can follow a number of different PDFs (probability density functions). This could be, for example, Gaussian, multinomial [10], or complement [13], and a Naive Bayes algorithm implementation exists for each of them.

For Gaussian, the distribution of events being described follow a normal curve. This uses the mean and standard deviation as descriptions of the normal curve. The multinomial distribution models the probability associated with n independent trials of a k-category event taking place and resulting in one of the k values for each trial. For complement, it is a NB classifier implmentation suited to imbalanced datasets. When there is a bias in the datasets, the probability of the occurance of the event not belonging to that category (i.e. belonging to all the other categories) is calculated. Then the smallest of those values is chosen because its complement would imply it is the highest chance of belonging to that category.

The previous article [2] demonstrated KNN (k-nearest neighbors), DT (decision trees), and RF (random forests) as algorithms for regression problems. The same three have alternate implementations for classification problems and are known as classifiers. The details of KNN, DT, and RF are discussed in [2] but they can be used, just like the Naive Bayes algorithms to do classification and compare how they perform.

Bagging and Boosting

The algorithms for RF discussed so far fall into the category of Bagging (bootstrap aggregating) algorithms. An alternate grouping is Boosting algorithms. [14] Two boosting algorithms that will be demonstrated are AdaBoost (Adaptive Boosting) and XGBoost (Extreme Gradiant Boosting).

Generally, both are examples of ensemble learning. This is where multiple models are run and the best answer from their collective answers is used. They differ in how the models are run, how the outputs are collected and processed and finally how they produce the final prediction. These are usable in both regression and classification problems. The idea is to reduce the variance from running individual models by combining the results of several different models or instances of a model to give a more stable answer.

In bagging:

- The series of weak learning models learn independently in parallel.
- It aides in reducing overfitting.
- It uses row sampling with replacement method for training subsets.
- The ensemble classifier counts the maximal number of class selection and assigns that answer.
- Suited to high variance.

In boosting:

- The series of weak learning models learn serially and adapt results from previous models to improve learning.
- The models are built serially and previous errors in the models are corrected subsequently.
- Higher weights are given to incorrect answers and lower weights to correct answers.
- Subsequent models learn from the assigned weights and focus on improving higher weighted models.
- The process terminates when maximal models is reached or dataset is correctly predicted.
- The final weighted majority answer is the output classification.
- Suited to high bias.

Different algorithms implement methods to accommodate the above behaviors. The dataset characteristics will determine which algorithm is best suited for training the model. Both approaches aim to improve prediction performance and must be tried to see how they perform on the dataset.

Analysis

Dataset

As with all machine learning problems, the dataset to analyze must be examined before applying any algorithms to build models. The familiar Boston housing dataset has been used to predict median home sale price. The median home sale price is a continuous random target variable value best suited to regression models.

The classifiers used to build classification models will attempt to predict a discrete random target variable. All the algorithms discussed in the theory section are used on this new dataset to see how they perform. The dataset used here is from [5] and consists of features describing potential pulsars. Some characteristics of the data include what is listed in the column headers below. Since the CSV file of data has no headers, they must be set in the Pandas dataframe created from reading in the CSV file using the traditional methods seen before. This allows for listing all the features shown.

In [27]:
```
import pandas as pd
import numpy as np
pulsar_df = pd.read_csv('HTRU_2.csv', header=None)
pulsar_df.columns = [['Mean of integrated profile', 'Standard deviation of integrated profile',
                'Excess kurtosis of integrated profile', 'Skewness of integrated profile',
                'Mean of DM-SNR curve', 'Standard deviation of DM-SNR curve',
                'Excess kurtosis of DM-SNR curve', 'Skewness of DM-SNR curve', 'Class' ]]
pulsar_df.head()
```

Out[27]:

	Mean of integrated profile	Standard deviation of integrated profile	Excess kurtosis of integrated profile	Skewness of integrated profile	Mean of DM-SNR curve	Standard deviation of DM-SNR curve	Excess kurtosis of DM-SNR curve	Skewness of DM-SNR curve	Class
0	140.562500	55.683782	-0.234571	-0.699648	3.199833	19.110426	7.975532	74.242225	0
1	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	0
2	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	0
3	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	0
4	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	0

Important things to note in the description of the data is that the features are real numbers. The last column is 'class', the target value column. The class says if the features described indicate the sample is a pulsar or not. The class is a discrete value and takes on the binary values of 0 (it is not a pulsar) and 1 (it is a pulsar).

In [30]:
```
pulsar_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17898 entries, 0 to 17897
Data columns (total 9 columns):
 #   Column                                        Non-Null Count  Dtype  
---  ------                                        --------------  -----  
 0   (Mean of integrated profile,)                 17898 non-null  float64
 1   (Standard deviation of integrated profile,)   17898 non-null  float64
 2   (Excess kurtosis of integrated profile,)      17898 non-null  float64
 3   (Skewness of integrated profile,)             17898 non-null  float64
 4   (Mean of DM-SNR curve,)                       17898 non-null  float64
 5   (Standard deviation of DM-SNR curve,)         17898 non-null  float64
 6   (Excess kurtosis of DM-SNR curve,)            17898 non-null  float64
 7   (Skewness of DM-SNR curve,)                   17898 non-null  float64
 8   (Class,)                                      17898 non-null  int64  
dtypes: float64(8), int64(1)
memory usage: 1.2 MB
```

The info command for the dataframe gives a basic count of the samples, including the quantity of pulsars found. Each of the columns is fully populated and do not have empty values. The overall description of the data is given below to show the total number of entries.

In [33]:
```
pulsar_df.describe()
```

Out[33]:

	Mean of integrated profile	Standard deviation of integrated profile	Excess kurtosis of integrated profile	Skewness of integrated profile	Mean of DM-SNR curve	Standard deviation of DM-SNR curve	Excess kurtosis of DM-SNR curve	Skewness of DM-SNR curve	1
count	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	
mean	111.079968	46.549532	0.477857	1.770279	12.614400	26.326515	8.303556	104.857709	
std	25.652935	6.843189	1.064040	6.167913	29.472897	19.470572	4.506092	106.514540	
min	5.812500	24.772042	-1.876011	-1.791886	0.213211	7.370432	-3.139270	-1.976976	
25%	100.929688	42.376018	0.027098	-0.188572	1.923077	14.437332	5.781506	34.960504	
50%	115.078125	46.947479	0.223240	0.198710	2.801839	18.461316	8.433515	83.064556	
75%	127.085938	51.023202	0.473325	0.927783	5.464256	28.428104	10.702959	139.309330	
max	192.617188	98.778911	8.069522	68.101622	223.392141	110.642211	34.539844	1191.000837	

In [35]:
```
totalCount = pulsar_df.Class.count()
print(f'total count of potential pulsars {totalCount}')
count = (pulsar_df['Class'] ==0).sum()
print(f'non-pulsars {count}')
count = (pulsar_df['Class'] ==1).sum()
print(f'pulsars {count}')
print(f'percentage of pulsars {count/totalCount}')
```

```
total count of potential pulsars Class    17898
dtype: int64
non-pulsars Class    16259
dtype: int64
pulsars Class    1639
dtype: int64
percentage of pulsars Class    0.091574
dtype: float64
```

The above statistics show some basic information about the quantity of pulsars involved. This shows that the data is mostly 0 and not 1. This can have an impact on a model's performance. Different algorithms will perform differently on the dataset. Some could just guess a value of 0 and be considered good, because the vast amount of data is 0 and not 1. This is where the evaluation of the model's performance is important because it will determine the quality of the predictions it generates in the context of very skewed data. The imbalance in the data set can result in selecting an algorithm that is best suited to the

distribution of the data.

Quality Measures

In order to judge the quality of the model's predictions, the outputs must be evaluated using a confusion matrix and classification report. These two concepts were originally defined and described in [1]. One goal of the confusion matrix is to describe the predictions made by the model on the test set. The model will be correct and incorrect on some of its predictions. Depending on what the dataset is describing, these errors in prediction could cause serious problems.

Assume a binary outcome dataset is being examined for prediction. Recall that when the model predicts something happened as true, when it did not actually occur in the test set, then the model has produced a false positive (this is also known as a Type I error). When the model predicts something happened as false, when it did actually occur in the test set, then the model has produced a false negative (Type II error). These kinds of statistics can be found in the confusion matrix to judge how well the model (and algorithm selected) is performing. The classification report can be similarly used to produce different scores. These include precision (how low the false positives were), recall (how correct the positives were) and F1-scores (a weighted average of the two for comparing the performance of the precision and recall amongst different models). All these statistics can be used in the evaluation and optimization steps to improve the predictive strength of the model.

In the confusion matrix, the output needs to be examined to account for how the data is organized. Typically, the confusion matrix lists the positive conditions for the predictions in rows and ground truth values in the columns.

The sklearn library outputs the values in the opposite order, namely the 0s (negative values) are listed first and the 1's (positive values) are listed second. This is done in both the rows of predicted conditions and the columns of ground truth conditions. This will be important in the classification report as well.

The classification report shows 0's (negative values) on the first row and then 1's (positive values) on the second row.

In [ ]:
```
predicted conditions                 true Conditions
         |                  (0)condiction neg   (1) condition pos <- total population
Predicted Condition Neg(0) [[ A=True Negative,  B=False Negative],
Predicted Condition Pos(1)  [ C=False Positive, D=True Positive]]
```

So for precision, of the true positives (or true negatives), the following operations would be done with respect to the data that appears in the confusion matrix and makes up the classification report.

Remember, the rows are the 'prediction' values and the columns are the 'ground truth' values. Using the definition of Precision = true positives divided by all the positive predictions (D/D+B). Recall = true positives divided by the total positive labels (D/C+D). Similarly for the negatives (0's), the Precision = A/(A+C), and the Recall = (A/A+B).

Experimental Results

This section examines the utilization of several different models described above. The overall performance of the models is examined as well as optimizations with an alternate class of algorithms. The pulsar data extracted above is used in the model training.

Functions

A few support functions [12] are introduced to facilitate running several different models and evaluating each of them. These support functions will help run different models with cross validation to generate scores to determine how well the model is performing. Additionally, as mentioned earlier, the confusion matrix and classification report are used to see the quality of the results produced from each algorithm.

These two functions are slightly modified from the original source in order to generically pass in the predictors (features) and response (target) variables so they are operated on inside the functions. These are shown next prior to being used to generate results.

In [37]:
```
# classifier execution function.  Takes 3 parameters
# The first is the model object for the specific type of model used
# The X is the predictor values matrix
# The y is the response values column vector
def clf_model(model, X, y):
    # pull in the libraries of interest
    from sklearn.model_selection import cross_val_score
    clf = model
    scores = cross_val_score(clf, X, y)
    print('Scores:', scores)
    print('Mean score:', scores.mean())
```

In [39]:
```
# confusion matrix and classification report
# The first is the model object for the specific type of model used
```

```
# The X is the predictor values matrix
# The y is the response values column vector
# The training and test data split happens with a fixed 25% for each use of the function.
def confusion(model, X, y):
    from sklearn.metrics import classification_report
    from sklearn.metrics import confusion_matrix
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
    clf = model
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
    return clf
```

Performance

In [42]:
```
# The data frame from earlier is first split up into the
# predictors (first 7 columns) and responses (last column)
# for all rows, based on the description give above.
X = pulsar_df.iloc[:,0:8]
y = pulsar_df.iloc[:,8]
```

In [44]:
```
# logistic regression.
# It does need to account for convergence time and hence the
# max number of iterations is set in the model used.
from sklearn.linear_model import LogisticRegression
clf_model(LogisticRegression(max_iter=1000, random_state=0), X, y)
```

```
Scores: [0.97486034 0.97988827 0.98184358 0.97736798 0.9782062 ]
Mean score: 0.9784332723007113
```

In [46]:
```
# Gaussian naive bayes
# Assumes independent and identically distributed input features and
# assumes they follow the same distribution, in this case, Gaussian.
# It could be multinomial or complement for two additional available models.
from sklearn.naive_bayes import GaussianNB
clf_model(GaussianNB(), X, y)
```

```
Scores: [0.96061453 0.92374302 0.94273743 0.92847164 0.96451523]
Mean score: 0.9440163679814436
```

In [48]:
```
# KNN
from sklearn.neighbors import KNeighborsClassifier
clf_model(KNeighborsClassifier(), X, y)
```

```
Scores: [0.96955307 0.96927374 0.97318436 0.9706622  0.97289746]
Mean score: 0.9711141653437728
```

In [50]:
```
# Decision Trees
from sklearn.tree import DecisionTreeClassifier
clf_model(DecisionTreeClassifier(random_state=0), X, y)
```

```
Scores: [0.96843575 0.96424581 0.96871508 0.96227997 0.96954457]
Mean score: 0.9666442360073738
```

In [52]:
```
# Random Forest
from sklearn.ensemble import RandomForestClassifier
clf_model(RandomForestClassifier(random_state=0), X, y)
```

```
Scores: [0.97709497 0.98324022 0.98072626 0.97485331 0.97848561]
Mean score: 0.978880074800083
```

All the models perform very well. They show a mean score (RMSE) ranging from 94%-97%. So these models are doing very well on this dataset. Even examining the individual scores of the cross validation operations over 5 folds, shows high scores. The earlier discussion about the heavy skew of most of the data being non-pulsars helps explain why they are performing as such. The additional evaluation techniques of confusion matrix and classification reports must be examined next.

In [54]:
```
# confusion matrix and classification report quality measures for each is generated.
```

In [56]:
```
confusion(LogisticRegression(max_iter=1000), X, y)
```

```
[[4095   20]
 [  63  297]]
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      4115
           1       0.94      0.82      0.88       360

    accuracy                           0.98      4475
   macro avg       0.96      0.91      0.93      4475
weighted avg       0.98      0.98      0.98      4475
```

Out[56]:

```
▾       LogisticRegression

LogisticRegression(max_iter=1000)
```

In [60]:

```
confusion(KNeighborsClassifier(), X, y)
```

```
[[4077   38]
 [  69  291]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      4115
           1       0.88      0.81      0.84       360

    accuracy                           0.98      4475
   macro avg       0.93      0.90      0.92      4475
weighted avg       0.98      0.98      0.98      4475
```

Out[60]:

```
▾ KNeighborsClassifier

KNeighborsClassifier()
```

In [62]:

```
confusion(DecisionTreeClassifier(random_state=0), X, y)
```

```
[[4047   68]
 [  60  300]]
              precision    recall  f1-score   support

           0       0.99      0.98      0.98      4115
           1       0.82      0.83      0.82       360

    accuracy                           0.97      4475
   macro avg       0.90      0.91      0.90      4475
weighted avg       0.97      0.97      0.97      4475
```

Out[62]:

```
▾       DecisionTreeClassifier

DecisionTreeClassifier(random_state=0)
```

In [64]:
```
confusion(GaussianNB(), X, y)
```

```
[[3946  169]
 [  52  308]]
              precision    recall  f1-score   support

           0       0.99      0.96      0.97      4115
           1       0.65      0.86      0.74       360

    accuracy                           0.95      4475
   macro avg       0.82      0.91      0.85      4475
weighted avg       0.96      0.95      0.95      4475
```

Out[64]:

▾ GaussianNB

GaussianNB()

In [68]:
```
confusion(RandomForestClassifier(random_state=0), X, y)
```

```
[[4095   20]
 [  59  301]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      4115
           1       0.94      0.84      0.88       360

    accuracy                           0.98      4475
   macro avg       0.96      0.92      0.94      4475
weighted avg       0.98      0.98      0.98      4475
```

Out[68]:

▾ RandomForestClassifier

RandomForestClassifier(random_state=0)

For the models run, the RandomForestClassifer shows very good performance in recognition of true pulsars. This is seen in the "1" row (positive indication of a pulsar) giving precision, recall, and the f1-score (the average of precision and recall scores) values that are better than the others. Logistic regression f1-score was the same, but its recall for positives was a little lower.

The Random Forest is an ensemble algorithm that makes use of many decision trees. So it is not surprising that it performs

better than the decision tree model and the other models. Additionally, there are more algorithms that can be used to try and increase the performance using this idea of applying improvements from running multiple models into subsequent models. That idea was originally discussed in the theory subsection on bagging and boosting.

Optimization

Boosting algorithms, like Adaboost and XGBoost, can also be used on the pulsar dataset to see how well they perform in comparison with the previous classifiers. The same functions used above are exercised to produce mean scores, confusion matrix, and classification reports. Here are both examples showing their performance.

In [95]:
```
from sklearn.ensemble import AdaBoostClassifier
clf_model(AdaBoostClassifier(), X, y)
```

```
Scores: [0.97430168 0.97988827 0.98128492 0.97597094 0.97708857]
Mean score: 0.977706874833175
```

In [97]:
```
confusion(AdaBoostClassifier(), X, y)
```

```
[[4094   21]
 [  63  297]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      4115
           1       0.93      0.82      0.88       360

    accuracy                           0.98      4475
   macro avg       0.96      0.91      0.93      4475
weighted avg       0.98      0.98      0.98      4475
```

Out[97]:

▾ AdaBoostClassifier

AdaBoostClassifier()

In [99]:
```
# The XGBoost requires installation first.
import sys
!{sys.executable} -m pip install xgboost
from xgboost import XGBClassifier
```

```
Collecting xgboost
  Downloading xgboost-3.0.0-py3-none-manylinux_2_28_x86_64.whl (253.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 253.9/253.9 MB 734.6 kB/s eta 0:00:00m eta 0:00:01[36m0:00:01
Requirement already satisfied: scipy in /home/ap/anaconda3/lib/python3.10/site-packages (from xgboost) (1.1
0.0)
Collecting nvidia-nccl-cu12
  Downloading nvidia_nccl_cu12-2.26.2.post1-py3-none-manylinux2014_x86_64.manylinux_2_17_x86_64.whl (291.7
MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 291.7/291.7 MB 1.4 MB/s eta 0:00:00m eta 0:00:01[36m0:00:01
Requirement already satisfied: numpy in /home/ap/anaconda3/lib/python3.10/site-packages (from xgboost) (1.2
3.5)
Installing collected packages: nvidia-nccl-cu12, xgboost
Successfully installed nvidia-nccl-cu12-2.26.2.post1 xgboost-3.0.0
```

In [101…
```
from xgboost import XGBClassifier
clf_model(XGBClassifier(), X, y)
```

```
Scores: [0.97653631 0.98128492 0.9801676  0.97513272 0.97680916]
Mean score: 0.9779861420046485
```

In [103…
```
confusion(XGBClassifier(), X, y)
```

```
[[4084   31]
 [  53  307]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      4115
           1       0.91      0.85      0.88       360

    accuracy                           0.98      4475
   macro avg       0.95      0.92      0.93      4475
weighted avg       0.98      0.98      0.98      4475
```

Out[103…

```
▾                                XGBClassifier
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              feature_weights=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
```

The overall performance of these two models is seen in their respective outputs. The XGB appears to perform better with slightly improved score for recall.

Conclusions

This article has covered a lot of material with respect to how classifiers work. The goal in classification of predicting discrete target values can be achieved through a number of different algorithms. Classic logistic regression is nearly always the first one used. Other algorithms like KNN, DT, RF all have classifer implementations that can be applied as well.

The introduction to Naive Bayes as a new model relies on assumed independance and distributions of features. With those assumptions, NB offers a family of algorithms that perform reasonably well in prediction.

Lastly, optimization approaches for classification included introducing boosting algorithms to build models for prediction. The boosters provide equivalent or better performance in comparison to the earlier classifiers.

The overall quality of prediction was judged using confusion matrix and classification report. The classification report contains a handful of metrics like precision and accuracy that can give better insight into how the model is performing on its prediction task over the test dataset. These tools prove to provide a very good indication about the performance of the models.

References

[1] Patel, Amit. EDA and Visualization Part Two. https://ap20.github.io/nnj/NL/edutechrev/EDAandVisPart2_apatel.html

[2] https://ap20.github.io/nnj/NL/edutechrev/IntroMLPart1_regression_apatel.html

[3] https://en.wikipedia.org/wiki/Pulsar

[4] R. J. Lyon, B. W. Stappers, S. Cooper, J. M. Brooke, J. D. Knowles, Fifty Years of Pulsar Candidate Selection: From simple filters to a new principled real-time classification approach MNRAS, 2016.

[5] R. Lyon. "HTRU2," UCI Machine Learning Repository, 2015. [Online]. Available: https://doi.org/10.24432/C5DK6R. Downloaded from https://archive.ics.uci.edu/dataset/372/htru2.

[6] https://www.kaggle.com/datasets/altavish/boston-housing-dataset

[7] https://en.wikipedia.org/wiki/Logistic_regression

[8] https://en.wikipedia.org/wiki/Sigmoid_function

[9] https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[10] https://www.geeksforgeeks.org/multinomial-naive-bayes/

[11] The Python Workshop, Second Edition. https://www.amazon.com/dp/1804610615

[12] The Python Workshop, source code: https://www.packtpub.com/product/the-python-workshop-second-edition/9781804610619?utm_source=github&utm_medium=repository&utm_campaign=9781804610619

[13] https://www.geeksforgeeks.org/complement-naive-bayes-cnb-algorithm/

[14] https://www.geeksforgeeks.org/bagging-vs-boosting-in-machine-learning/

Appendix

Boston Housing Values

This appendix examines using the boosting algorithms on the Boston median house price regression problem. The two boosting algorithms examined earlier have regressor versions (instead of the classifiers used for pulsar data) that can be used on continuous valued target random variables.

First the dataset must be read into a dataframe and then a helper function (modified slightly from the original supplied in the source code) is used to run the model via cross validation using the number of folds specified. This produces the same measures of RMSE for the cross validation folds and the overall mean to determine model quality. The two boosting models can then be compared to see how they perform.

In [114…
```
housing_df = pd.read_csv('HousingData.csv')
# clean up the data
housing_df = housing_df.dropna()
# declare X and y
housing_X = housing_df.iloc[:,:-1]
housing_y = housing_df.iloc[:, -1]
```

In [116…
```
# Define a helper function to use
# The model is passed in, and the prediction matrix variable X and response column vector y
def regression_model_cv(model, X, y, k=5):
    from sklearn.metrics import mean_squared_error
    from sklearn.model_selection import cross_val_score

    scores = cross_val_score(model, X, y, scoring='neg_mean_squared_error', cv=k)
    rmse = np.sqrt(-scores)
    print('Reg rmse:', rmse)
    print('Reg mean:', rmse.mean())
```

In [118…
```
from sklearn.ensemble import AdaBoostRegressor
regression_model_cv(AdaBoostRegressor(), housing_X, housing_y)
```

```
Reg rmse: [3.48826519 3.44866114 5.51712808 6.51074081 4.0870298 ]
Reg mean: 4.61036500627967
```

In [120…
```
from xgboost import XGBRegressor
regression_model_cv(XGBRegressor(), housing_X, housing_y)
```

```
Reg rmse: [3.11376639 5.36460979 6.15243328 6.64486279 4.12696925]
Reg mean: 5.080528301575429
```

This shows the performance is relatively strong compared to the other regressors used on this dataset in the earlier article [1].