Introduction to Machine Learning Algorithms - Part One: Regression

Introduction

This article is focused on starting the basic introduction to machine learning algorithms. Previous articles in the series [1] [2] have focussed on the basics of exploratory data analysis and visualization. Those articles covered how to clean and manage the datasets in preparation for their use in machine learning. Those articles also covered the fact that some machine learning algorithms fall into solving two board categories of problems: linear and logistic regression. Where the former's purpose is to predict a target value for a continuous random variable and the latter for a discrete random variable (binary or fix set of classes).

The last article [2] described a series of ten steps in the process of doing machine learning. Those earlier articles covered the first 6 steps, but the remaining steps 7-10 must be taken to go from a well-groomed dataset to training and prediction. The exercises demonstrated in this article will cover:

1] Setting the algorithm to use, which could have hyperparameters to tune it. Once the algorithm is fixed, then it leads to creating and training a model. 2] The dataset is used to train a model, and the model is used to make predictions. 3] The results of the prediction must be evalulated to determine the quality of performance. 4] Lastly is optimization, which can be done to improve the quality. This is typically done using trail and error, grid search, or tuning hyperparameters.

All of these topics will be the focus of this article as well as demonstrating sample software that is capable of doing each of these on candidate datasets.
As was the case with previous articles, this one will similarly explore well known public datasets. Furthermore, the book [3] used to guide through these algorithms provides additional details beyond the scope of this summary article. The examples and software [4] are copied here for the reader to give them an opportunity to learn and digest these concepts independently. Readers are encouraged to reproduce these examples from the book in their own machine learning development environment for self-practice. This will help to build familiarity with the concepts, how to apply them, and how to reason and draw conclusions about the datasets shown or new datasets they analyze.

Theory

In the case of a continuous random varible, there are a number of different regression approaches that can be used. Linear, Ridge, and Lasso are examples discussed. This class of algorithms is know as regressors. Similarly, for another problem category in machine learning, the task of classification is to predict a discrete variable (label) target value. These are known as classifiers.

In addition to the three above, KNN (k-nearest neighbors), decision trees, and random forests are additional classes of algorithms that can be used for regression problems. The random forests are a collection of decision trees that are used in a model to produce an 'ensemble' answer for the prediction. The regressor versions of their implementations will be examined to compare how they perform on a dataset. Their classifier versions will be used in other labeling problem datasets.

Generally, linear, Ridge and Lasso are attempting to fit a line (or polynomial) to a set of data points. The idea in regression being to minimize the distance between the actual data point values and the line, which represents the prediction of the target variable based on the samples provided. The idea of minimizing the distance is the notion of measuring how good the model's predicted value is from the target value. Typically, mean squared error (MSE) is used, but there could be other measures.

Training a model with a dataset can be problematic if the model's predictions are too closely tied to the data values the model was trained on. This is an case of overfitting. The previous article [2] discussed using the train_test_split (and possibly validation dataset) function to partition the full data into training dataset for the model and test dataset for determining how well the model is performing using some measure. To compensate for overfitting, the idea of regularlization is introduced.

Ridge and Lasso are two algorithms that can be used because they act on the linear model's coefficients (or weights of each term in the matrix of data value columns). These algorithms apply a penalty term to affect the weight. For Ridge, it is based on the L2 penalty (Euclidean distance) and for Lasso, is it based on the L1 penalty ('taxicab' or 'manhattan' distance).

The KNN algorithm's purpose is to determine the value of the unknown target variable for a sample entry based on the output values of those examples that are closest to this sample entry. The number of examples are considered neighbors of the sample and the algorithm attempts to utilize some number (k) of them in determining the sample's target variable value.

Typically, the definition of closest is some measure, like distance, that is used to determine which features line up most closely with the sample entry.
Then those k nearest neighbor's target variable values are averaged to produce the sample's missing target variable value. The value of how many neighbors k to use when running the algorithm can be optimized.

Decision trees and a collection (ensemble) of decision trees, known as random forests, are another set of algorithms that can be used for regression and classification. Decision trees use a binary partition strategy to bifurcate on features to reach the target variable value to predict. This happens when the binary decisions can not go any further and the tree reaches a leaf which is the predicted value for the target variable.

Analysis

Recall in the introduction, the steps outlined included evaluation of the predictions the models were making and optimizing them for higher quality. Now that a set of regressor algorithms are known to utilize in building models (linear, Ridge, Lasso, KNN, Decision Trees, Random Forests), some consideration must be given to fine tuning their performance. This is done through tuning hyperparameters for each algorithm or using better model building techniques.

One such concept that must be considered in linear problems beyond regularization is cross-validation to "evaluate" the algorithm selected, model trained and the predictions it is making. Cross-validation basically processes different subsets (or folds) of the training data. The model is trained multiple times on different subsets of the training data to generate mean test scores. By having multiple test scores based on the number of folds created (3, 5, 10) from the input dataset, the model's ability to predict is better reflected in these scores to determine its overall accuracy.

When discussing KNN, the number of neighbors k to consider was configurable.
It is a hyperparameter that can be tuned by training the model with different values of k to see what produces the best quality answers. The selected number of k neighbors should produce optimal prediction results from the model.

There are hyperparameters associated with random forests as well that can be tuned when training different models to see which produce the best quality predictions. Some of the tunable parameters include the number of trees in the forest, the maximum depth of the tree, the maximum number of features to consider when splitting, the mininum number of samples required before a split is made, and the minimum number of samples at the leaves [3].

Just as with linear regression, the idea of cross-validation can be applied to these algorithms to determine the quality of their predictive performance.

Many times the exact numbers to use for these parameters are not know apriori. This requires doing a search of the space (range of values) that the parameter values can be set to. This is referred to as grid search. There are several built-in methods to achieve this operation to find the right parameter values.

Sometimes, the grid search applies the different hyperparameters to the algorithm and produces an answer. Other times the search looks for the parameters and they have to be plugged into the algorithm and a model trained to see how it performs.

Cross validation and grid search will be utilized in running these algorithms to produce models of different predictive quality. The answers generated will include a number of results from building and running the models over different sets of data or with different hyperparameter values. From the set of answers produced by the models, the quality measure will yield which one is best.

Experimental Results

The discussion about linear regression algorithms and other regressors will now be demonstrated. The well known canonical Boston Housing dataset used in first article [1] will be used to see how these algorithms perform in constructing models that can predict the target variable (median home value) based on set of housing features used to train the models.

As mentioned earlier, linear regression uses mean squared error as a measure of how well it is performing in predicting the target variable's value. This is found from partitioning the original dataset into training and test sets and then using the test set values in the model (data it has not seen during the training step) to compare against the predictions made by the model. Cross validation can also be done to see how well the model performs on different folds of the dataset.

To accomplish this, two functions are defined to be used to quickly run multiple iterations with different algorithms to determine how well those models are performing. The functions are defined as below.

Linear Regression

This section introduces linear regression and techniques to utilize with those algorithms.

In [11]:
```python
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
```

In [12]:
```python
def regression_model(model):
    # Create training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
    # Create the regressor: reg_all
    reg_all = model
    # Fit the regressor to the training data
    reg_all.fit(X_train, y_train)
    # Predict on the test data : y_pred
    y_pred = reg_all.predict(X_test)
    # Compute and print RMSE
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    print('RMSE: {}'.format(rmse))
```

In [13]:
```
def regression_model_cv(model, k=5):
    '''
    k = number of folds of the dataset to utilize.  Default is set to 5 folds.
    This value of k is a hyperparameter that can be changed and optimized.
    '''
    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())
```

Now use the Boston Housing dataset from a csv file. Read the data in, clean it up by removing null values, and then split out the predictor columns (the matrix X of features) and the target column (the column vector y of target values). In this case, the target variable is the median house price. This data can be used in the two functions to execute and produce rmse values to measure the accuracy of the predictions.

In the case of cross-validation, the function takes the number of folds of the dataset to use to train the model. Different values of this hyperparameter can be executed to give an idea of what kind of training works best.

In [15]:
```
# Read in dataset file
housing_df = pd.read_csv('HousingData.csv')
# display the beginning entries
housing_df.head()
```

Out[15]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

In [16]:
```
# clean up the data a little to get rid of empty values
housing_df = housing_df.dropna()
# Setup the X matrix of predictor features and y target column vector
# get all the columns except the last one
X = housing_df.iloc[:,:-1]
# get the last column (containing the median house price, the target variable to predict
y = housing_df.iloc[:,-1]
# run the function a few times to get different rmse values for predicting the target value
regression_model(LinearRegression())
regression_model(LinearRegression())
regression_model(LinearRegression())
regression_model(LinearRegression())
```

```
RMSE: 4.866591953081344
RMSE: 4.44382262973528
RMSE: 4.16957311171683
RMSE: 5.022213861683472
```

Different values are produced each time the function is called because the train_test_split function is not producing the exact same values each time it is called. The results show the values produced are fairly close based on the RMSE of the predicted values and actual values found in the test set.

Now, examine the cross-validation execution using a few different values of k. For k=3,5,6. The function executes and produces the regression's RMSE based on the number of k folds passed into the function. Then the average of those runs is taken to produce the final answer.

In [18]:
```
regression_model_cv(LinearRegression(), k=3)
regression_model_cv(LinearRegression(), k=5)
regression_model_cv(LinearRegression(), k=6)
```

```
Reg rmse:  [ 3.72504914  6.01655701 23.20863933]
Reg mean:  10.983415161090855
Reg rmse:  [3.26123843 4.42712448 5.66151114 8.09493087 5.24453989]
Reg mean:  5.337868962878355
Reg rmse:  [3.23879491 3.97041949 5.58329663 3.92861033 9.88399671 3.91442679]
Reg mean:  5.086590810801078
```

Regularization

This section examines the effects of regularization to mitigate overfitting by using Ridge and Lasso. These are two algorithms available in the sklearn library and just like LinearRegression() used earlier, these can be passed into the functions created to yield (for default k value of 5) RMSE values and the mean.

In [27]:
```python
from sklearn.linear_model import Ridge, Lasso
regression_model_cv(Ridge())
regression_model_cv(Lasso())
```

```
Reg rmse:  [3.17202127 4.54972372 5.36604368 8.03715216 5.03988501]
Reg mean:  5.232965166251771
Reg rmse:  [3.52318747 5.70083491 7.82318757 6.9878025  3.97229348]
Reg mean:  5.60146118538429
```

For Ridge, the overall mean is not better than k=6 folds of the dataset using the LinearRegression model. But the spread of individual values is smaller (when comparing with the k=5 folds example because this execution uses the same k=5 folds by default), hence reducing overfitting. This is more apparent in the k=3 folds case which manifests wider range of values. The Lasso case is not that much better because of the L1 distance metric being used. The penalty imposed by the L1 distance is not as effective.

KNN

Now lets examine the KNN and decision trees algorithms to see how they perform. As was already discussed earlier, the idea behind KNN is to utilize the target variable values of adjacent examples to determine the value of the new data point's target variable value. This involves using the KNN regressor and adjusting the hyperparameter k (number of neighbors). The default number of neighbors that KNN uses if not supplied is 5. This is the case with the first call below.

The k number of neighbors can also be supplied as an input parameter to see which one gives the best performance. This is exercised over different subsets of the data by using the same cross validation function as before. The cross-validation function still uses the default value of 5 folds (defined in the function definition above) of the dataset provided.

```
In [32]: from sklearn.neighbors import KNeighborsRegressor
         regression_model_cv(KNeighborsRegressor())
         regression_model_cv(KNeighborsRegressor(n_neighbors=4))
         regression_model_cv(KNeighborsRegressor(n_neighbors=7))
         regression_model_cv(KNeighborsRegressor(n_neighbors=10))
```

```
Reg rmse:  [ 8.24568226  8.81322798 10.58043836  8.85643441  5.98100069]
Reg mean:  8.495356738515685
Reg rmse:  [ 8.44659788  8.99814547 10.97170231  8.86647969  5.72114135]
Reg mean:  8.600813339223432
Reg rmse:  [ 7.99710601  8.68309183 10.66332898  8.90261573  5.51032355]
Reg mean:  8.351293217401393
Reg rmse:  [ 7.47549287  8.62914556 10.69543822  8.91330686  6.52982222]
Reg mean:  8.448641147609868
```

Grid search is a better way of trying to find the correct number of neighbors compared to manually entering different values for n_neighbors. It is a built-in module that allows for systematically searching the space of possible values that neighbors can take on and attempts to find the optimal one. The following function captures the operations of interest to conduct the grid search with cross-validation.

```
In [35]: from sklearn.model_selection import GridSearchCV
         def regression_model_cv_gridsearch(model):
             neighbors = np.linspace(1,20,20)
             k=neighbors.astype(int)
             param_grid = {'n_neighbors':k}
             model_tuned = GridSearchCV(model, param_grid, cv=5, scoring='neg_mean_squared_error')
             model_tuned.fit(X,y)
             k = model_tuned.best_params_
             print("Best n_neighbors: {}".format(k))
             score = model_tuned.best_score_
             rsm = np.sqrt(-score)
             print("Best score: {}".format(rsm))
         # run the function with the knn as the model
         regression_model_cv_gridsearch(KNeighborsRegressor())
```

```
Best n_neighbors: {'n_neighbors': 7}
Best score: 8.516767055977628
```

This shows that grid search is able to tune the hyperparameter and find its optimal value. In this case it was 7 neighbors. This score compares similar to what was obtained when using the cross-validation function above and supplying different neighbor count values into the calls.

Decision Trees

The next two sections attempt to utilize a different kind of algorithm. This section describes how to use decision trees described earlier to find a leaf value that represents the target variable value. The following section describes how an ensemble of decision trees make up a random forest and the kind of accuracy that random forests can supply.

The hyperparameters for both algorithms were given earlier. The methods below allow for optimizing them by using cross-validation during the training and for a search operation for finding the best hyperparameter values. Both have simple regressors that can be used to produce baseline results.

```
In [39]: from sklearn import tree
         from sklearn.ensemble import RandomForestRegressor
         regression_model_cv(tree.DecisionTreeRegressor(random_state=0))
         regression_model_cv(RandomForestRegressor(random_state=0))
```

```
Reg rmse:  [3.7647936  7.26184759 7.78346186 6.48142428 4.79234165]
Reg mean:  6.016773796161434
Reg rmse:  [3.21859405 3.76199072 4.96431026 6.55950671 3.7700697 ]
Reg mean:  4.454894289804201
```

This shows the second result for the forest produces a better answer than the first result from using a single decision tree.

Random Forests

This last example is to demonstrate the hyperparameter tuning that can be done with Random Forests. This tuning allows for calculating the best parameters to use to get the best score.

The parameter grid is supplied to the search function and it gives the range of values to test over. Changing the minimum to maximum values in the grid changes the search space. As before, the negative MSE measure is used, and the resulting set of parameters produced is shown as well as the overall score.

```
In [44]: from sklearn.model_selection import RandomizedSearchCV
         param_grid = {'max_depth': [None,1,2,3,4,5,6,8,10,15,20],
                       'min_samples_split':[2,3,4,5,6],
                       'min_samples_leaf':[1,2,3,4,6,8],
                       'max_features':[1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4]}
         reg = RandomForestRegressor(n_jobs=-1, random_state=0)
         reg_tuned = RandomizedSearchCV(reg, param_grid, cv=5, scoring='neg_mean_squared_error', random_state=0)
         reg_tuned.fit(X,y)
         p = reg_tuned.best_params_
         print("Best params: {}".format(p))
         score = reg_tuned.best_score_
         rsm = np.sqrt(-score)
         print("Best score: {}".format(rsm))
```

```
Best params: {'min_samples_split': 5, 'min_samples_leaf': 2, 'max_features': 0.7, 'max_depth': 10}
Best score: 4.46557417781969
```

Next is increasing the total number of trees to use in the algorithm. Here the estimators is set to 500. This represents the number of decision trees in the forest, and hence the ensemble of values, that will be generated and run to train the model.

In [47]:
```
regression_model_cv(RandomForestRegressor(n_jobs=-1, n_estimators=500))
```

```
Reg rmse:  [3.21198483 3.7475531  4.72724717 6.47405632 3.8740773 ]
Reg mean:  4.4069837453959
```

Lastly, set the grid values calculated above producing the best score with the new number of esimators to give a new answer.

In [50]:
```
regression_model_cv(RandomForestRegressor(n_jobs=-1, n_estimators=500,
                                          random_state=0, min_samples_split=5,
                                          min_samples_leaf=2, max_features=0.7,
                                          max_depth=10))
```

```
Reg rmse:  [3.18498898 3.59234342 4.66618434 6.43013587 3.81099639]
Reg mean:  4.336929799775126
```

Notice the different values for score produced and how the final one with optimized parameters produces the best answer.

Conclusions

This article gave a very quick tour of some algorithms that can be used for regression problems. This is the class of ML problems that required predicting a continuous random variable value for the target variable of interest. It started with the classic linear regression algorithms to train some models. To combat the problems of overfitting, Ridge and Lasso regularization techniques were introduced.

Then the additional algorithms of KNN, decision trees and random forest were demonstrated for regression. These algorithms implement both regressor and classifier versions. To begin finding the best scores requires a number of additional techniques and hyperparameter tuning. The other techniques involved cross validation to train models across more variations and interations of the training set. It also included using randomized search and grid search to cycle through different values in the range of values a hyperparameter can take on in order to find the hyperparameter value that gives the best performance.

References

[1] Patel, Amit. EDA and Visualization Part One. https://ap20.github.io/nnj/NL/edutechrev/ExploreDAandVisPart1_apatel.html

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

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

[4] 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