Hyperparameter Optimization

Introduction:

For this exercise I experimented with both the GridSearchCV and BayesSearchCV model selectors on a couple of ML algorithms. I decided on k-nearest neighbors and random forest and compared the accuracy of the models from the default results, GridSearchCV optimized and BayesSearchCV optimized.

Dataset Description:

I used the wine quality dataset[1] for this and picked the white wine dataset because it has more entries than the red dataset. This data includes 11 features and a target value called "quality" that is between 0 and 10.

[1]: https://archive-beta.ics.uci.edu/dataset/186/wine+quality

Experimental Setup:

For the data setup, I went with the white wine as mentioned above, then split the test data to be 20% of the overall data and for the models I went with k nearest neighbors and random forest classifiers, knn was picked because I wanted to play around with the hyperparameters on this model more and random forest was picked because it was the most accurate in the previous exercise.

Accuracy this time is recorded using scikit-learn's accuracy_score function that exists for classification models.

For this experiment I used scikit-learn's GridSearchCV and BayesSearchCV model selectors which I gave a model, a list of hyperparameters, a scoring system and cross validation fold count to determine out of the list of parameters I give it, which ones are the best. BayesSearchCV I experimented with a little bit and reduced it's number of iterations because it was testing with hyperparameters it had already tested on.

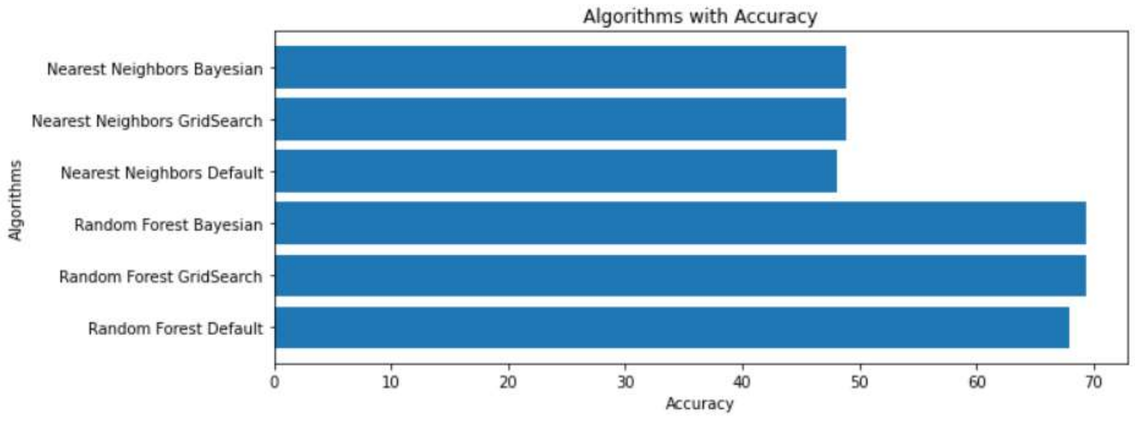
I later noticed that the Bayesian optimization on random forest was running faster so I decided to time all of the models and output that too.

I then would graph the results and make a list of the algorithms sorted by accuracy[2] then by time.

[2]: code for that inspired by https://stackoverflow.com/questions/48053979/print-2-lists-side-by-side user SCB

Results:

```
Accuracy Score for  Random Forest Default 67.9591836734694 %
Accuracy Score for  Random Forest GridSearch 69.38775510204081 %
Accuracy Score for  Random Forest Bayesian 69.38775510204081 %
Accuracy Score for  Nearest Neighbors Default 48.06122448979592 %
Accuracy Score for  Nearest Neighbors GridSearch 48.87755102040816 %
Accuracy Score for  Nearest Neighbors Bayesian 48.87755102040816 %
```



```
List of algorithms sorted best to worst:

Random Forest Bayesian: 69.38776% accuracy, 17.37181 seconds
Random Forest GridSearch: 69.38776% accuracy, 28.46992 seconds
Random Forest Default: 67.95918% accuracy, 0.84377 seconds
Nearest Neighbors GridSearch: 48.87755% accuracy, 4.71129 seconds
Nearest Neighbors Bayesian: 48.87755% accuracy, 6.26871 seconds
Nearest Neighbors Default: 48.06122% accuracy, 0.03603 seconds
Best Algorithm by accuracy and time is: Random Forest Bayesian
```

This image contains the output of my program, as we can see the gridsearch and bayessearch performed the same with what it was given and both outperformed default hyperparameters. Though one thing to note is that the bayessearch ran much faster than the gridsearch for random forest and about as fast for the nearest neighbors. From this, I can see with a more complex problem or larger dataset the Bayesian optimization may be better to use given how much faster it runs for a similar result.

They only ended up with small improvements, likely due to how simple the problem was and how good the default values are for the problem.

Code:

In the repository under main.py.