Machine Learning Project 1

Author One TBAIRD7@JHU.EDU
Tommy Baird
Johns Hopkins University
Baltimore, MD 21205, USA

Editor: Tommy Baird

Abstract

In this paper, we will discuss the application of a k-nearest neighbors machine learning algorithm to try and predict the outcomes of both classification and continuous targets. This application will be tested on 6 different data sets of varying length and complexity. In all but 1, we will show significant performance improvement over a null model predictor.

Keywords: Machine Leaning, K Nearest Neighbors

1 Introduction

As mentioned in the abstract, we will look to make predictions on 6 different data sets using the machine learning algorithm of k-nearest neighbors. The implementation is relatively straightforward in the fact that we will simply compare normalized features that define each data point, find the k other data points that are closest in value, and then make our prediction based on those neighbors. We do not have any previous expertise in the areas dissected in these data sets, so we are assuming that all features are relevant and equally weighted unless otherwise guided to dismiss.

1.1 Pre-Processing Steps

For all 6 data sets, there were a variety of data cleaning techniques that were employed before beginning each run of the algorithm. In most cases, we transformed all numbers into floats to allow for easy normalization and comparisons. For the normalization, we used the z-score normalization process to provide an equal comparison across different features leveraging different measurement units. In most other cases, we used boolean values within one-hot encoding to represent our non-numeric features. We have a few other cases in which we leveraged integers specifically which we will discuss in more detail within their specific test overviews.

1.2 Experimental Approach

Our approach to this class was in most cases scripted from the project. We tuned our hyper-parameters which were labeled as p, the exponent of our Minkowski metric, k, the number of neighbors to consider, e, the error term when considering the range of "correct" estimates while condensing our regression sets, and s, the multiplication factor for our kernel when we multiply by 1 divided by the product of s and the standard deviation of our test set

All data sets provided at `https://archive.ics.uci.edu/`.

targets. The cross validation was the defined approach in the project documentation and condensed nearest neighbor was our selection of two condensing techniques provided. The random search tuning method on the other hand was one of many possible. We selected it with consideration to efficiency and time management.

1.2.1 5x2 Cross Validation

We employed a 5x2 cross validation technique to tune and test our data. For tuning, we split off 20% of the original data set to use for tuning. The other 80% will then be used for testing. Each of our 5 runs will consist of splitting the test set in half (stratifying where possible to ensure equal distributions), shrinking down the set using a condensed nearest neighbor approach to help performance, and then testing the tuning set against each. This will allow us to test sets of hyper-parameters 10 times on 10 potentially unique condensed test sets.

1.2.2 Condensed Nearest Neighbor

As mentioned, condensed nearest neighbor was employed to help performance on the tests themselves. We first start with a random data point from our full test set. We will then iterate through the test set to see if we can get a correct prediction when looking at the single nearest neighbor within our new condensed set. If that prediction is wrong, then we add the data point to our test set as it will provide detail that does not currently exist in the set. We iterate through our test set repeatedly until we have a full iteration where no new data points are added.

1.2.3 Random Search Hyper-Parameter Tuning

For tuning, we implemented a random selection technique to test hyper-parameters of the test set. Our other consideration was a full grid search which would have allowed us to test all possible input combinations, but there are time constraints for this project that must be incorporated into our assumptions. Random search within our set of parameter combinations was presented as a viable alternative during office hours. As an example within the regression test sets, performing 20 different hyper-parameter tuning tests on the computer hardware test set took less than an hour which was completely reasonable. On the other hand, the Abalone test set took around 12 hours to perform one 5x2 hyper-parameter test. Some of these time constraints could be enhanced by fine tuning the code to improve performance or perhaps by providing better parameters to test on, but within the bounds of the project time limits, we will make due with what we can.

2 Breast Cancer

This test set was relatively straight forward which made it a good proving ground for our initial runs of our algorithms. There were only 2 classes for our algorithm to choose, and as shown below, there is a decent number of data points for each to allow the system to find plenty of relevant neighbors as it parses through the features:

Class	Count
2	444
4	239

For tuning, please note that the e and s terms are not relevant here nor for any of the classification sets. I also want to highlight again that I leveraged a random search to pick hyper parameters to tune on. In this case, we picked 5 random combinations of a possible 8.

As we tuned both our p and our k, it did not appear that there was much of a difference in the values that we chose. It did seem to prefer a higher k so that the algorithm would be able to avoid noise from false neighbor assignments, but given that tuneID of 2 with a k of 3 performed worse than tuneID 4 with a k of 1, it does not seem to always be the case that a higher k leads to higher results. The same could be said for our p value because while we will chose the highest possible value in p of 2, there is two tuneIDs of 3 and 1 that outperformed tuneID of 4 by using a p of 1. Given all of this, we will move forward with our tests with a p of 2 and a k of 4. These are both the max possible values for each.

TuneID	p	k	e	s	AveragePerformance	TestsRun
7	2	4	1	1	0.959	10
3	1	4	1	1	0.955	10
1	1	2	1	1	0.942	10
4	2	1	1	1	0.939	10
2	1	3	1	1	0.903	10

Below is the output of our test leveraging our tuned hyper parameters. Overall, it seems to have a very solid success rate given that the average is over 95%. I would also note that given the fact that there are more data points with a class of 2, it does make sense for us to have a slightly better success rate with that class assignment.

actualValue	correctAssignment%	totalTestCases
2	96.958	1775
4	94.136	955

Finally, we can do our comparison to the Null model to prove that our algorithm performed better than simply choosing the most likely output. We can see that we outperformed significantly which further proves our optimism towards the success of the test.

TestOutput	NullModelOutput
95.971	65.018

3 Car Evaluation

We will argue that this set is not the best data set for a k nearest neighbor algorithm. As shown below, there is a heavy skew to unacc classes. There is a decent number of acc as well, but only a few good and vgood for which we can look for:

Class	Count
unacc	1210
acc	384
good	69
vgood	65

To further this point, we can see the general distribution of each feature value split up by the one hot encoding that was leveraged for this set. If we look at two features as an example, we see that something arbitrary like the number of doors that a car have does not appear to give much of a clue to the evaluation of the car itself. I would argue that we would potentially remove this feature from future tests.

Class	doors_2	doors_3	doors_4	doors_5more
acc	0.211	0.258	0.266	0.266
good	0.217	0.261	0.261	0.261
unacc	0.269	0.248	0.241	0.241
vgood	0.154	0.231	0.308	0.308

On the other hand, the safety feature does seem to show more meaningful results with the safety of the relevant vehicle given that we can see unacc having a larger percentage within the low safety bucket and vgood showing only results for high safety.

Class	safety_low	safety_med	safety_high
acc	0.000	0.469	0.531
good	0.000	0.565	0.435
unacc	0.476	0.295	0.229
vgood	0.000	0.000	1.000

As we move on to the tuning analysis, we can see below that there was not much deviance in performance as we moved around p and k. All of the results were somewhere between 81 and 83% success rate. We will choose the highest for now.

TuneID	p	k	e	s	AveragePerformance	TestsRun
2	1	16	1	1	0.832	10
4	2	14	1	1	0.829	10
1	1	8	1	1	0.824	10
5	2	6	1	1	0.823	10
3	2	12	1	1	0.814	10

Below is a breakdown of the results by class. As expected, unacc performed the best given its large percentage of constituents within the original data set. On the other hand, good and vgood performed horribly which we would expect given the few data points that we had to work with as neighbor examples.

actualValue	correctAssignment%	totalTestCases
acc	58.567	1535
good	21.091	275
unacc	93.926	4840
vgood	28.077	260

Finally, we can do our comparison to the Null model. We can see that while our performance was not as strong as we had hoped, we did outperform by more than 10% relative to the Null model.

TestOutput	NullModelOutput
80.695	70.043

4 Congress Voting

This data set seems ripe for a k nearest neighbor algorithm given how strictly politicians vote based on their party lines. The dividing line may not have been so defined in 1984 when these votes take place, but the performance of our algorithm performed very well which was helped by a decent distribution across the two classes:

Class	Count
democrat	267
republican	168

As we iterated through the data of this specific set, we had a choice to make on data features that included an abstain vote. It is hard to know the circumstances on each time that a person decides to not vote on a given issue, so we made an assumption that an abstain should be counted as its own form of vote. In other words, we updated "Yes" to be 1, "No" to be -1, and "?" which was the placeholder for abstain to be a 0. It could also have been not counted, but for a group of people whose job it is to vote on such matters, an abstain has to count for something.

Looking at our tuning, there was not any major deviances in how each performed. In hindsight, it may have been interesting to crank up the k value to help sort through the noise of the set and let the strong party lines sort each member into their respective class.

TuneID	p	k	e	s	AveragePerformance	TestsRun
6	2	12	1	1	0.903	10
2	1	12	1	1	0.886	10
5	2	11	1	1	0.876	10
0	1	10	1	1	0.875	10
7	2	13	1	1	0.872	10

To further the thought from above as we learn more about the algorithm, the below performance of the system by party would give us more inclination to increase k. Democrats were the larger party in size within the data set, so they had less chance of picking up noise from their rival party.

actualValue	correctAssignment%	totalTestCases
democrat	95.514	1605
republican	80.498	1005

However, even with tuning regrets, the model performed well as proven by its comparison to the Null model with half the mean squared error. See below for final results.

TestOutput	NullModelOutput
88.851	61.494

5 Abalone

This test case would have been an interesting one to test given its relatively defined classes.

	Class
count	4177.000
mean	9.934
std	3.224
min	1.000
10%	6.000
30%	8.000
50%	9.000
70%	11.000
90%	14.000
max	29.000

However, it became more of a lesson on the time restrictions that this algorithm brings. If given more time, it would have been a good test to see where optimization within the code could occur, or perhaps there were certain parameters that we could tune to speed things up. Unfortunately, it took approximately 12 hours to test out a single set of parameters. The lesson learned here is to start with the data sets that are the largest if you expect any meaningful tuning. The performance of that one test shown below:

TuneID	p	k	e	s	AveragePerformance	TestsRun
13	2	30	2	2	5.107	10

Amidst this strife, we still were able to perform well relative to the null model though. See below for results.

	Test Error	Null Error	Test SE	Null SE
mean	1.675	2.356	5.415	10.390
25%	0.601	0.925	0.361	0.857
50%	1.275	1.925	1.625	3.708
75%	2.205	3.075	4.863	9.453

6 Computer Hardware

We will see that this test case is a difficult one to apply to a k nearest neighbor approach. There is a relatively large distribution on our range of values with a decent sized standard deviation and skew towards lower performing values. Any noise from a too small or too large outlier could swing the regression estimate meaningfully.

	Class
count	209.000
mean	105.622
std	160.831
min	6.000
10%	17.000
30%	32.000
50%	50.000
70%	82.400
90%	261.000
max	1150.000

Looking at tuning, we were balancing the values of k to see if a larger number would help to offset outliers or if a smaller number would prevent them altogether. The below shows pretty clearly that a smaller k provides a better estimate. Note the below only shows our top 5 and bottom 5 tuning results to explain point above but also to preserve space.

TuneID	p	k	e	s	AveragePerformance	TestsRun
0	1	5	5	1	9760.836	10
1	1	5	5	2	9918.756	10
4	1	5	15	1	10145.173	10
6	1	7	5	1	10892.277	10
8	1	7	10	1	10985.879	10
27	2	7	10	2	12693.981	10
31	2	10	5	2	13274.678	10
30	2	10	5	1	13357.415	10
14	1	10	10	1	13424.203	10
34	2	10	15	1	13657.382	10

Using the best performing parameters, we created a model that performed relatively well. Our standard error grew as the range of possible values disperses with higher performing hardware. We outperformed the null model as shown in the average of the standard error columns, and we also outperformed at each percentile which proves the consistency of the model throughout the test set.

	Test Error	Null Error	Test SE	Null SE
mean	35.994	88.560	8771.767	24007.752
25%	5.910	40.820	34.935	1666.302
50%	12.812	66.820	164.140	4464.960
75%	33.296	82.820	1108.646	6859.212

7 Forest Fires

This data set appears to be a more extreme version of the previous in terms of its skew. As shown below, a significant chunk of our targets are at or near zero.

	Class
count	517.000
mean	1.111
std	1.398
min	0.000
10%	0.000
30%	0.000
50%	0.419
70%	1.729
90%	3.268
max	6.996

As we worked through tuning our parameters, it seems as if a smaller p and a larger k performed best. This is a deviance from the narrative that we saw with computer hardware as less k improved performance there. In this case, the best way to control that skewed data was by including more values. It also shows that a p of 1 performs best along with a pairing of a lower e and a higher s.

TuneID	p	k	e	s	AveragePerformance	TestsRun
19	1	16	0.500	2	2.581	10
21	1	16	1.000	2	2.615	10
9	1	8	1.000	2	2.681	10
12	1	12	0.500	1	2.690	10
37	2	12	0.500	2	2.691	10
33	2	8	1.000	2	2.695	10
41	2	12	2.000	2	2.702	10
24	2	4	0.500	1	2.771	10
35	2	8	2.000	2	2.848	10
32	2	8	1.000	1	2.890	10

Looking at our results below of the test, and it appears that our expected values were too high for smaller values and too small for larger values. While a larger k did seem to perform better, I wonder if including that many neighbors resulted in a less exact result hovering generally around 1 in most cases. As shown in the mean of our standard error columns, the null outperformed our model here.

	Test Error	Null Error	Test SE	Null SE
mean	1.158	1.123	1.892	1.802
25%	0.667	0.830	0.445	0.689
50%	1.091	1.091	1.190	1.190
75%	1.474	1.091	2.172	1.190

8 Conclusions

The k-nearest neighbor algorithm performed relatively well to its rival null model. The null model does feel like a straw man comparison in most cases, but it does give an indication that the model itself is leveraging data to provide an informed decision. While we assume most machine learning algorithms to be slow moving at times, hopefully future implementations allow for the experiments to pare down seemingly irrelevant data points to speed up the calculations. As noted within the car evaluation section, the safety metric seemed much more informational than the number of doors of the car. Weighing both the same or even weighing the doors at all seems misguided. This would improved the aforementioned time constraints as well as the quality of our distance metric to define the neighbors.

In terms of the individual test cases, the algorithm seemed to have varying results. It performed well when we had a somewhat balanced target set with plenty of data points in each resulting class. This proved true with the breast cancer and congress voting data sets. On the other hand, it did not seem to like when we had imbalanced sets or skewed data as we saw with the car evaluation and the forest fire data sets. There were some inconsistencies on whether we should pick a larger or smaller k to make up for that skew, but with the right time and understanding of the data at hand, we probably could tune this algorithm to be a better evaluator of its closest neighbors.

Acknowledgments and Disclosure of Funding

This project was completed through the guidance of Dr. John Sheppard during the Introduction to Machine Learning Class within the Masters of Computer Science Program at Johns Hopkins University. There is no external funding for the creation, maintenance, and application of these tests.