Empirical Comparison of Standard Machine Learning Algorithms with a Time Complexity-Reduced K-NN Algorithm

Sebastian Kleinerman sekleine@ucsd.edu

Adam Casper acasper@ucsd.edu

March 24, 2018

Department of Cognitive Science, University of California, San Diego, La Jolla, CA 92093

Abstract:

Our objective was to analyse 3 pre-built Sklearn algorithms on 3 different datasets and compare them to a KNN(K nearest neighbors) algorithm we built from scratch. Typically, KNN algorithms are computationally expensive due to having to calculating the distance from the sample point to every point in the training set. Our "KNN-bubble" algorithm instead creates a "bubble" or space around the sample point and then looks at just the points captured inside the bubble, and then uses those points for classification of the sample point. The Sklearn algorithms we used for comparison were Decision Tree, SVM, and SciKit-Learn's KNN. In addition, we compared runtime with a simple generic KNN algorithm we wrote from scratch since Sklearn probably uses some optimization techniques that aren't related to the core algorithm. We found that SciKit-Learn's KNN drastically out-performed ours in terms of runtime, however, the simple generic KNN was comparable.

Introduction:

Many machine learning algorithms are very computationally expensive. Work can be done to allow algorithms with inefficient time-complexities to run faster with better processors and other technology. However, another solution is simply optimizing an algorithm in order to reduce the time complexity it requires it to run successfully.

The goal of this experiment is to show that better models can be written and utilized without expensive computational resources, in order to increase efficiency. Of course, requiring nearly perfect accuracy might present an issue, but in some cases, a faster algorithm with a decent accuracy could be a life-saver. This paper investigate if a less tedious version of KNN classification can produce reliable results and good accuracies at a quick pace.

We'll call this new KNN "bubbleKNN" and this is how it works:

1. A sample point is chosen, one at a time, from the data set (as it is done in the generic KNN).

2. We create a region or “bubble” around this data point. However, given that every dataset are going to have differently spaced datapoint, first we need to find, on average, how far these data points are away from each other. To do this we first grab a small number of data points of the same class and calculate, then average their distances from each other. Then we can use multiples of this distance as a hyperparameter to decide the bubble size around our data point.
3. Then we take k samples from within our bubble and see which of their classes is most numerous.
4. Then as in generic KNN the we assign the sample point to the class that is most numerous.

PSEUDOCODE:

Determining bubble radius:

```
For training set: first n samples
        If of the same class: determine and average distance=avgdist
For sample in Testset:
```

Testing points in bubble:

```
        For feature in sample:
                If  (feature - avgdist*multiplier) < Trainfeature < (feature + avgdist*multiplier)
                   Match++
                If Match++ = len(sample)
                    Bubblesamples.append(Trainfeature)
                    Increment num_in_bubble
               Else
                     Break from loop
               If num_in_bubble = k
                     Break from loop
      For  points in Bubblesamples
           Check class label:
          If classlabel_1 > classlabel_0
                Sample = classlabel1
          Else:
                Sample = classlabel0
        Predictlist.append(classlabel)
```

Methods:

Multiple libraries were used in the process of building this project in order to produce and display the desired results. The SciKit-Learn library was used due to its popularity in the world of Data Science as well as its range of usable features and the ease of use it provides.

The supervised learning algorithms that were tested include Decision Tree, K-Nearest Neighbors, SVM, and the newly built version of K-Nearest Neighbors (bubbleKNN). Three algorithms belong to the SK-Learn library, and the bubbleKNN and a generic KNN algorithm was written and implemented from scratch. The Decision Tree models were tested on heights of 1, 2, 3, 4, and 5. The regular version of K-Nearest Neighbors was tested using 1, 2, 3, 4, and 5 neighbors (K). The SVM parameters were "linear" and "rbf" for the kernel, "gamma" values of 1e^-7, 1e^-6, 1e^-5, and 1e^-4, and finally the values tested to avoid misclassifying each training sample were 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0 (larger C values are a smaller margin, and vice-versa).

The datasets used all come from the University of California, Irvine repository. They are titled, "Auto MPG", "Covertype", and "Letter Recognition." For each dataset, each model was trained on different ratios of the full set. The first train to test ratio was 20:80, the second was 50:50 and the third and final ratio was 80:20. So, for example, the first ratio resulted in training the datasets on 20% of the entire set, and then tested on the other 80%.

The "Auto MPG" dataset was used in order to classify cars between being American-made or foreign. In order to do this, an extra column was added which only contains either a 0 or 1. A 1 is found whenever a car is American-made, and 0 otherwise. The "Letter Recognition" dataset was used in order to simply classify whether a given letter belonged to the first or second half of the alphabet. If the letter was 'M' or before 'M', then the binary value would be a 1. If the letter was anything after 'M', then the binary value used to classify it was a 0. Finally, the "Covertype" dataset was used in order to see if the given tree type was either part of the first group made up of "Spruce/Fir", "Lodgepole Pine", and "Ponderosa Pine", or the second group which is made up of "Cottonwood/Willow", "Aspen", "Douglas-fir", and "Krummholz." If the given tree was part of the first group, then it was labeled a 1, and if it was part of the second, then it was labeled a 0.

Initially, the "Auto MPG" dataset was made up of 398 instances and 8 attributes, but after cleaning, it got reduced to 392 instances due to missing features in some of the data points. The 8th attribute was replaced from a string value of it's make/model to a binary classification value (1 or 0). The "Letter Recognition" dataset was made up of 20,000 instances and 16 attributes. We used 1,000 instances due to time constraints and being run on a laptop computer. The last attribute was changed to a binary classification related to belonging to the first or second half of the alphabet as mentioned above. Finally, the "Covertype" dataset started out with 581,012 instances and 54 attributes. We again reduced the

instances to 1,000. The last attribute became a binary classification value related to tree cover type as mentioned above. Also 2 attributes that were “one-hot encoded” were dropped, because we believed that these would be over represented using the 3 classifier types.

This program was ran using Python 3.6.1 in a Jupyter Notebook of version 4.3.0 on macOS Sierra version 10.12.6. The computer used for testing is a MacBook Pro 15” from 2016 with a 2.7 GHz Intel i7 processor and 16GB 2133MHz LPDDR3 RAM. This laptop represents a very decent computer that a student or a new Data Scientist could be using. It has a 4 core processor. Furthermore, other libraries were used in the process of making this project. These include Pandas (version 0.20.1), NumPy (version 1.12.1), and Matplotlib (version 2.0.2). The most important of these libraries is NumPy since it allowed for data storing and manipulation using arrays. Pandas dataframes were very useful for data purposes of data loading, cleaning, and visualization.

After each data set was cleaned and made up of entirely scalar values operated on them in the following ways. First we used cross validation to determine the best hyper-parameters for each classifier. Then we split the data into training and test partitions using the ratios described above. Then for each split we ran the classifiers on them each 3 times and averaged their performance accuracies.
Then, we populated a pandas dataframe with the accuracy scores from each classifier.

Results:

We found that, aside from the letter data set, we were able to achieve decent classification accuracies for all classifiers (appx 75-90%). The letter data set classification was not much better than chance at around 50-60% accuracy.

All three KNN classifiers had similar accuracy. When we tested speed we found that the SKlearn KNN classifier was significantly faster compared to the generic built KNN and bubbleKNN at appx 2-3 ms when ran for 100 loops. The generic built KNN classifier was faster than the bubbleKNN but not dramatically, at appx 40-50 ms to 90-100 ms respectively.

Training Results

Out[23]:

AccuracyType(Train%)(Dataset)	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
trainAccuracy(.2)(Auto_MPG) #1	0.987179	0.807692	0.987179	0.641026
trainAccuracy(.2)(Auto_MPG) #2	1.000000	1.000000	1.000000	0.730769
trainAccuracy(.2)(Auto_MPG) #3	1.000000	0.871795	1.000000	0.858974
trainAccuracy(.2)(Cover_Type) #1	0.980000	1.000000	0.830000	0.630000
trainAccuracy(.2)(Cover_Type) #2	0.870000	1.000000	0.740000	0.580000
trainAccuracy(.2)(Cover_Type) #3	0.890000	1.000000	0.910000	0.580000
trainAccuracy(.2)(Letter_Recognition) #1	0.980000	1.000000	1.000000	0.870000
trainAccuracy(.2)(Letter_Recognition) #2	1.000000	1.000000	1.000000	0.980000
trainAccuracy(.2)(Letter_Recognition) #3	0.950000	1.000000	0.920000	0.810000
trainAccuracy(.5)(Auto_MPG) #1	0.994898	0.877551	0.994898	0.826531
trainAccuracy(.5)(Auto_MPG) #2	0.984694	1.000000	0.984694	0.729592
trainAccuracy(.5)(Auto_MPG) #3	1.000000	0.903061	1.000000	0.790816
trainAccuracy(.5)(Cover_Type) #1	0.676000	1.000000	0.752000	0.644000
trainAccuracy(.5)(Cover_Type) #2	0.780000	1.000000	0.740000	0.620000
trainAccuracy(.5)(Cover_Type) #3	0.796000	1.000000	0.724000	0.652000
trainAccuracy(.5)(Letter_Recognition) #1	0.992000	0.972000	1.000000	0.936000
trainAccuracy(.5)(Letter_Recognition) #2	0.992000	1.000000	1.000000	0.908000
trainAccuracy(.5)(Letter_Recognition) #3	1.000000	1.000000	1.000000	0.952000
trainAccuracy(.8)(Auto_MPG) #1	0.990415	1.000000	0.990415	0.769968
trainAccuracy(.8)(Auto_MPG) #2	0.993610	0.884984	0.993610	0.725240
trainAccuracy(.8)(Auto_MPG) #3	0.990415	0.884984	0.990415	0.792332
trainAccuracy(.8)(Cover_Type) #1	0.802500	1.000000	0.752500	0.695000
trainAccuracy(.8)(Cover_Type) #2	0.807500	1.000000	0.762500	0.710000
trainAccuracy(.8)(Cover_Type) #3	0.752500	1.000000	0.742500	0.735000
trainAccuracy(.8)(Letter_Recognition) #1	0.995000	1.000000	1.000000	0.942500
trainAccuracy(.8)(Letter_Recognition) #2	0.990000	1.000000	1.000000	0.900000
trainAccuracy(.8)(Letter_Recognition) #3	0.992500	1.000000	1.000000	0.927500
trainAccAverage(.2)(Auto_MPG)	0.995726	0.893162	0.995726	0.743590
trainAccAverage(.2)(Cover_Type)	0.913333	1.000000	0.826667	0.596667
trainAccAverage(.2)(Letter_Recognition)	0.976667	1.000000	0.973333	0.886667
trainAccAverage(.5)(Auto_MPG)	0.993197	0.926871	0.993197	0.782313
trainAccAverage(.5)(Cover_Type)	0.750667	1.000000	0.738667	0.638667
trainAccAverage(.5)(Letter_Recognition)	0.994667	0.990667	1.000000	0.932000
trainAccAverage(.8)(Auto_MPG)	0.991480	0.923323	0.991480	0.762513
trainAccAverage(.8)(Cover_Type)	0.787500	1.000000	0.752500	0.713333
trainAccAverage(.8)(Letter_Recognition)	0.992500	1.000000	1.000000	0.923333

Testing Results

Out[24]:

AccuracyType(Train%)(Dataset)	Decision_Tree	K-Nearest Neighbors	SVM	newK-Nearest Neighbors
testAccuracy(.2)(Auto_MPG) #1	0.993631	0.700637	0.993631	0.735669
testAccuracy(.2)(Auto_MPG) #2	0.990446	0.757962	0.952229	0.694268
testAccuracy(.2)(Auto_MPG) #3	0.990446	0.757962	0.984076	0.770701
testAccuracy(.2)(Cover_Type) #1	0.587500	0.680000	0.612500	0.585000
testAccuracy(.2)(Cover_Type) #2	0.617500	0.717500	0.610000	0.625000
testAccuracy(.2)(Cover_Type) #3	0.647500	0.715000	0.595000	0.637500
testAccuracy(.2)(Letter_Recognition) #1	0.882500	0.940000	0.830000	0.960000
testAccuracy(.2)(Letter_Recognition) #2	0.942500	0.952500	0.830000	0.932500
testAccuracy(.2)(Letter_Recognition) #3	0.867500	0.977500	0.902500	0.890000
testAccuracy(.5)(Auto_MPG) #1	0.989796	0.734694	0.989796	0.739796
testAccuracy(.5)(Auto_MPG) #2	1.000000	0.785714	1.000000	0.826531
testAccuracy(.5)(Auto_MPG) #3	0.984694	0.801020	0.984694	0.770408
testAccuracy(.5)(Cover_Type) #1	0.696000	0.764000	0.728000	0.688000
testAccuracy(.5)(Cover_Type) #2	0.640000	0.824000	0.736000	0.692000
testAccuracy(.5)(Cover_Type) #3	0.748000	0.748000	0.704000	0.668000
testAccuracy(.5)(Letter_Recognition) #1	0.972000	0.968000	0.852000	0.952000
testAccuracy(.5)(Letter_Recognition) #2	0.972000	0.980000	0.840000	0.960000
testAccuracy(.5)(Letter_Recognition) #3	0.976000	0.972000	0.840000	0.948000
testAccuracy(.8)(Auto_MPG) #1	1.000000	0.759494	1.000000	0.746835
testAccuracy(.8)(Auto_MPG) #2	0.987342	0.746835	0.987342	0.721519
testAccuracy(.8)(Auto_MPG) #3	1.000000	0.822785	1.000000	0.759494
testAccuracy(.8)(Cover_Type) #1	0.710000	0.780000	0.650000	0.670000
testAccuracy(.8)(Cover_Type) #2	0.660000	0.800000	0.710000	0.660000
trainAccuracy(.8)(Cover_Type) #3	0.690000	0.810000	0.760000	0.640000
testAccuracy(.8)(Letter_Recognition) #1	0.940000	0.980000	0.810000	0.950000
testAccuracy(.8)(Letter_Recognition) #2	0.970000	0.980000	0.890000	0.960000
testAccuracy(.8)(Letter_Recognition) #3	0.980000	0.960000	0.850000	0.930000
testAccAverage(.2)(Auto_MPG)	0.991507	0.738854	0.976645	0.733546
testAccAverage(.2)(Cover_Type)	0.617500	0.704167	0.605833	0.615833
testAccAverage(.2)(Letter_Recognition)	0.897500	0.956667	0.854167	0.927500
testAccAverage(.5)(Auto_MPG)	0.991497	0.773810	0.991497	0.778912
testAccAverage(.5)(Cover_Type)	0.694667	0.778667	0.722667	0.682667
testAccAverage(.5)(Letter_Recognition)	0.973333	0.973333	0.844000	0.953333
testAccAverage(.8)(Auto_MPG)	0.995781	0.776371	0.995781	0.742616
testAccAverage(.8)(Cover_Type)	0.686667	0.796667	0.706667	0.656667
testAccAverage(.8)(Letter_Recognition)	0.963333	0.973333	0.850000	0.946667

Conclusion:

In order to acquire a better understanding of how different machine learning algorithms work, it is crucial to apply them to different datasets and train/test ratios. As a result, observations can be made to notice similarities and differences among the resulting accuracies.

We expected the bubbleKNN algorithm to be faster due to the fact that not every sample point has to visit, and then calculate the distance, to every point in the training set. However, given that the accuracy was comparable we feel like it may be due to the implementation in code, rather than the number of operations. Also, we compared test times on our smallest dataset with the least number of instances. A decrease in runtime may improve as the number and complexity of the dataset increases. We are aware similar KNN algorithms most likely exist. However, we purposely did not research for similar versions of KNN because we wanted to remain relatively naive in creating our own algorithm in hopes we might stumble upon some new improvements. We feel further investigation of our algorithm and its implementation in code may improve its performance.

Bonus Points:

We believe some aspects of our project may warrant bonus points.

First and foremost we designed a machine learning classifier algorithm entirely from scratch without looking at other sources for guidance. We encountered many hurdles and spent many hours doing this, but we felt it would be more fruitful in possibly coming up with something novel if we isolated ourselves from similar work that has already been done in the past.

Secondly, we successfully programmed our algorithm and got it to perform on par with existing KNN algorithms. We did not use any external machine learning libraries our source code in doing so. We encountered many obstacles but we went back to the chalkboard each time. Many of the times we had to isolate chunks of code to try to see where we went wrong. We are both cogsci students and very novice programmers and this proved especially challenging at our skill level.

Thirdly, we had to modify a lot of optimization code or create them from scratch in order to tune our hyperparameters. This was also, a very tedious process.

While most students probably chose to utilize already existing machine learning algorithms we, instead, tried and succeeded to create our own.

Member Roles: *All work was done in a 50/50 coordinated effort by both members, Adam Casper and Sebastian Kleinerman*.

References

Caruana, Rich and Alexandru Niculescu-Mizil. “An Empirical Comparison of Supervised Learning Algorithms.” 23rd International Conference on Machine Learning. 25 June 2006.