Journal of Machine Learning Research 21 (2019) 1-10 Submitted 10/19; Published 11/19

Parameter Tuning with Variants of K Nearest Neighbor

Brandon Sladek BRANDONSLADEK@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Jared Thompson J.A.THOMPSON22@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Kyle Hagerman HAGERMANKYLE96@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Ryan Hansen RYANHANSEN2222@GMAIL.COM
School of Computing
Montana State University
Bozeman, MT 59718, USA

Abstract

The k-nearest neighbor algorithm is a method for predicting a class value for an unlabeled data point by assigning it the modal or average class value of its k nearest neighbors in the training set. In this project we examined the relative performance of five distinct nearest neighbor algorithms using six data sets and various parameter values for k. Four of these algorithms employ some kind of training set reduction, which decreases computational time for predicting data points by reducing the overall size of the training set prior to making predictions for test points. Despite clever clustering methods and training data reduction techniques, the basic k-nearest neighbor algorithm performed the best.

Keywords: K Nearest Neighbor, Clustering, Training Data Reduction, Parameter Tuning

1. Introduction

In this experiment we will compare the prediction performance of five different implementations of the k-nearest neighbor algorithm. The k-nearest neighbor algorithm is an extremely popular and widely used machine learning algorithm that predicts a test point's class value by examining the *k* nearest neighbors of the test data point, and assigns the mode or mean of the neighbors' class values as the class of the test value. We implemented a standard k-nearest neighbor implementation that uses the entire training set. The other four algorithms all employ a method of training set reduction that attempts to increase the prediction accuracy of the test set.

Two of the reduction algorithms, condensed and edited k-nearest neighbor, both iterate over the training set until a suitable set of data has been formed. The other two, k-means clustering and k-medoids clustering, reduce the data set into representative clusters that will be represented by single points used for classification.

All five algorithms have been implemented for both categorical and regression classification. We have three data sets that classify data into nominal class values, and three that are represented as regressions. By accommodating both types of predictions we can truly compare the effectiveness of each algorithm. We are using a separate loss function to analyze each classification type, 0/1 Loss Accuracy for categorical and Mean Squared Error for the regression data sets. The relative performance between the two types of predictions represented by the loss function analysis will be enough to compare the algorithms and determine their rankings with respect to one another.

2. Problem

The idea behind machine learning is to create a model that can classify incomplete data or make predictions given the presence of certain variables. Many algorithms have been suggested in the search for the best performance in prediction scenarios. This study will compare five such algorithms to determine the best use of their applications. Regression data sets can be quite difficult to fit a model to, and because of their different approaches, these algorithms must be tested to find which provides the best results for a given data set. By adding categorical data sets into the mix we may find a well-rounded algorithm or an algorithm that is particularly better or worse at one type of prediction (classification or regression). This will be useful information to anyone trying to classify similar data sets.

3. Application Architecture

The script is broken down into classes with each representing a piece of the work needed to implement a fully realized solution. In order of execution, the process follows these steps: data acquisition, data preprocessing, cross validation partitioning, execution of specified algorithms on specified data sets, results processing to calculate loss function values, and a final experiment runner script to execute the various parts of the code in order.

The data acquisition class, named DataApi, operates by pulling the data set information from the local csv files and creating a Pandas dataframe object for each data set. Put simply, a Pandas dataframe object is just a Python 2D array with some extra helper methods for accessing and analyzing the data given row and column indexes (McKinney (2010)). The DataApi class also has some helper methods needed by various parts of the application.

We implemented a KNN parent class from which all four remaining algorithm classes inherit. We used this class hierarchy relationship because some version of KNN classification is used by all five algorithms. The primary difference is not in the logic, but in the data passed in. The four classes that inherit from KNN (Edited KNN, Condensed KNN, K-Means, and K-Medoids) pass in alternative training sets built by their child-class-specific logic. For example, the Edited KNN class used the regular (parent-class) implementation of the k_nn() method, but ran the k_nn() method with an "edited" training set built within the ENN child class. In this way, each of the child classes handled any additional data preprocessing that needed to be done prior to running k_nn() to compute the test predictions given the training data.

In the Preprocessor class, we implemented a general function called preprocess_raw_data_frame(), which generically handles replacing/imputing missing values and normalizing each data set. It also decides whether or not it needs to normalize each column categorically or continuously dynamically.

The ExperimentRunner uses the DataApi, Preprocessor, and CrossValidator classes to call their respective methods. More specifically, when the run_experiment() method gets called, it fetches the data from the DataApi, preprocesses it by calling the preprocess_raw_data_frame() method in the Preprocessor class, partitions the data into 10 cross validation partitions, and then calls the corresponding algorithm class implementation entry point method to run each of the specified algorithms against each of the test sets that make up all the cross validation folds. The ParameterTuner class is simply used as a single source of parameters for each of the algorithms, fetched by calling the get_parameters() method and specifying the name of the algorithm for which we want the corresponding list of parameters to use in the current experiment. The algorithm implementation then returns a dictionary of predictions, which gets processed by the Results class using one of two loss functions, 0/1 Loss and Mean Squared Error, depending on the data set.

4. Preprocessing

Our first problem was differentiating between categorical and continuous data. As we chose Euclidean distance as the distance metric for KNN, we needed some way to quantify distance between categorical values. We chose to map each categorical value to a number, hopefully in an order that makes sense. This worked very well for the Car data, as the 'low','medium', and 'high' values were mapped to logically corresponding values 1, 2, and 3. This means low is closer to medium than it is to high, which makes sense. By doing it this way, though, we introduce a bias in other categorical data, like months, that may not be subject to the same kind of logic.

Second, we wanted each attribute to be meaningful and have equal representation when calculating the distance between two points. In Euclidean distance, if one attribute is on the scale of 10^{10} and another attribute is on the scale 10^{-10}, the latter will have almost no influence on the resulting distance calculation. We fixed this with normalization, where we map each value to the closed interval $[0, 1]$. This means each attribute has equal representation in the distance calculation, and additionally our mean squared error result should return a value between 0 and 1, as well, for helpful comparison.

5. Algorithms

5.1 K Nearest Neighbor

The primary algorithm for this project was KNN. We use it to predict class values for both classification and regression. Since all the algorithms ultimately run KNN after doing their respective preprocessing on the training data, the KNearestNeighbor class is a parent class that makes up the core of the logic for the various algorithm implementations.

5.2 Description

The k-nearest neighbor algorithm works for both the classification and regression data sets. Given a metric space, $M = (X, d)$, our data set, $D \subset M$, and a query point, $p \in M$, KNN returns the k points in D that return the smallest values for $d(p, y_i)$ where $y_i \in D$. For classification problems, a test point is classified based on the mode of the classes of the k nearest neighbors. Whereas for regression the class values are numbers, so our prediction for the class of a test point p is the mean of the class values of the k nearest neighbors.

5.3 Implementation

Because every algorithm implements some version of KNN, we chose to implement KNN as the parent class and the rest of the other algorithms as child classes. This design choice allowed the knn_predict() method to be implemented once in the parent KNN class, and then reused by each of the child algorithm classes, thereby reducing the coding overhead. Given the distance matrix for a dataframe, a training set, and a test set, the knn_predict() method iterates through the test set mapping each point to the corresponding row in the distance matrix (representing the distances from our test point to every point in the training set), and then finds the k smallest neighbor distances that match with points in the training set. Given the k nearest neighbors, a prediction is made based on the highest representation of classes in the neighbor set (the mode for classification or mean for regression), and a dictionary of predictions is returned containing a tuple holding the predicted class value and actual class value for each point in the test set.

5.4 Edited Nearest Neighbor

Edited Nearest Neighbor, or ENN, is a variant of k-nearest neighbor, with the only variation being extra preprocessing done on the training set before running the actual algorithm.

5.5 Description

In ENN, we first filter the training set points by running KNN against each individual training set point, against the rest of the training set points, and remove the training set points that were predicted incorrectly. By doing this, we end up with a new "edited" training set that is a subset of the original (full) training set. The edited training set contains all the points that were predicted correctly, using all the other training points as the training set to predict each individual training point.

5.6 Implementation

The EditedKNN class extends the KNN class, and mostly relies on the KNN class to do all the computations. Like all the other algorithm classes in this application, the EditedKNN class calls into the parent class KNN.knn_predict() method. However, when passing the training data and test data parameters, the EditedKNN knn_predict() call uses the training data as both the training set and the test set. This is because in ENN we wish to first filter the training set by predicting the class value for each training set point, and removing those that we classified incorrectly. So in this case, the training set is also the test set.

5.7 Condensed Nearest Neighbor

This algorithm reduces the training set by selecting cluster boundary case data points that are harder to classify. By capturing these edge cases that most clearly define the boundaries between classes, we will be able to predict "easier" points more readily.

5.8 Description

The Condensed Nearest Neighbor algorithm, or CNN, is designed to reduce the data set for KNN prediction by finding the list of nearest non-class neighbor for each point in the training set. This "reduced" training set is then ran through normal KNN prediction. The reasoning behind running the CNN algorithm is that it reduces the execution time for large KNN data sets, and the logic behind how to "smartly" reduce that data set is to return a list of the hardest points to classify. That list amounts to, essentially, the boundary points between clusters. If we can assume that classes are more likely to cluster together, the hardest points to classify in those clusters will be the points on the fringe; the edge cases, which will make up the training points in the CNN training set.

The logic behind CNN is to reduce the data set to those edge-case points, so that what is returned is a training set where the boundary cases between clusters are clearly defined. The hope, and also the major drawback of CNN, is that to work well it requires that our clusters are well separated and that the distances between clusters is relatively uniform. In this scenario CNN works well. Alternatively, if the clusters are not separated but all of them share the same distance region then CNN works, though probably not as accurately as KNN. The major failure of CNN happens if there is a possibility that a whole cluster of points with the same class value will never actually get represented within the training set. For example, if two clusters are close together (so much so that their boundaries actually bleed into one another), and the third cluster is far away, the training set of nearest non class neighbors will only be a representation of the two clusters close together. Hence, CNN has serious drawbacks in this case.

5.9 Implementation

CondensedKNN was implemented with respect to our entire project architecture. Meaning, the actual CNN child class takes in an unchanged training set, constructs the condensed CNN training set, and then calls the parent class KNN.knn_predict() method to make the KNN predictions for the test points using the condensed training set. To construct the condensed training set, the get_condensed_training_set() method (which implements most of logic in CNN) searches through every every row and every column of a 2D distance matrix made from the original training set. The distance matrix, is a square 2D array where each row represents a unique data point and every column value in that row pertains to the distance between that row point and every other data point in the data frame. By searching through this 2D matrix, we find the smallest not-class distance value to the row and add that to a list. We then update the original training set by removing every value not in the Z list, resulting in a smaller "condensed" training set.

5.10 K Means Clustering

The K-means clustering algorithm aims to cluster the data set into meaningful "summary" points, where each summary point represents the average point for a cluster of points.

5.11 Description

Starting with k random points, we iteratively calculate centroids of the subset of points closest to each random point. We then update the centroid for each cluster, and recalculate the associated cluster assignments for all the points. We continue doing this until either the centroids stop changing more than a threshold amount, or we reach our pre-specified maximum iterations count of 10. The output is k centroids, each of which represents a cluster of points with (hopefully) similar class values. We then use 1-KNN on the smaller, representative data set (just the centroids) to predict the class value for a given test point.

5.12 Implementation

From the training set we pick k random points to serve as the basis for forming clusters, the centroids. We then place every other training data point into a cluster based on its closest centroid. Once we make all the cluster assignments for an iteration, we average the values of each point in the cluster to form a new data point to serve as the centroid for the entire cluster in the subsequent iteration. We stop iterating when we have a sufficient set of cluster centroid points, at which point we call the parent class KNN.knn_predict() method to make predictions for all the test points using the cluster centroids as the training data.

5.13 K Medoids Clustering

This algorithm will cluster the training set into a set of data points that best represent k clusters in the data set. Each medoid is the best representative point for each cluster.

5.14 Description

By randomly selecting k data points (the medoids) from the training set, we can cluster the remaining data points into sets based upon which medoid is closest to each of them. Then, after we have generated clusters, we can find the best representative medoids from those clusters to use as our training data for making predictions using KNN.

5.15 Implementation

Upon receiving a training set and test set from the cross validation class, a distance matrix is calculated for the entire data frame. Then, we randomly pick indexes to represent the initial medoids. These medoids are used in tandem with the distance matrix to cluster each data point in the training set. Then with a triple-nested for loop, for each cluster we compare each data point to each other data point and generate a score to determine if a new medoid should be chosen. The score itself is a straight sum of the potential medoid to each other data point in the cluster. The data point with the lowest score is therefore the closest to each other data point and represents the center of the cluster. This list of medoid indices is then passed to the do_knn() method where we make predictions for the test data and run the loss functions on the results.

6. Experiment

This section discusses the experimental motivation and general approach for accepting or rejecting our hypothesis for the various data sets and potential k-values.

6.1 Approach

We had five different algorithms we wanted to compare using different data sets and parameters over the course of this experiment. We compared the performance of the various dataset-algorithm-parameter combinations through comparable loss function values.

Using Python and the Pandas dataframe library, we opted to run 10-fold class validation on each of the data sets except the CNN algorithm with the Abalone data set, where it was reduced to 5 to speed up the execution time. Cross validation helped to ensure our results were not particular to a specific training/test set. In probabilistic experiments, it is possible to get good or bad results due to random chance. Cross validation helps mitigate this problem by running the same process (in this case the algorithms) on various samples of the data. Finally, to ensure that each column attribute in our feature vector held equal weight, we normalized the column attributes to be values between zero and one.

Everything in the project was run through the ExperimentRunner class. We chose to run the experiment for each algorithm given a set of k values. For our k values, we chose the set [1, 4, 8]. The goal was to see how the various algorithms performed with different k values, i.e. watch the performance increase in the middle k-value region, and hope that it would plateau in the jump between 4 and 8.

6.2 Hypothesis

Our hypothesis was a ranking based on performance of each algorithm. We outline our ranking and justification in the table below.

	Performance Rank (1 best, 5 worst)
k-NN	**2** - aggregate of the whole data set (best with mid range k)
Edited k-NN	**1** - constructed for accuracy for the most points (best with higher k)
Condensed k-NN	**5** - constructed for identifying bad data (best with higher k)
k-means	**4** - prone to outliers, and no guarantee clusters form classes (best with low k)
k-medoids	**3** - step above k-means, outlier resistant (best with low k)

6.3 Results

The following is the output of our loss functions for each data set, algorithm and parameter value for k. A higher accuracy indicates better performance, and a higher mean squared error indicates worse performance. The following results were obtained by running on 3 k values - 1, 4, and 8. We chose these values because 8 seemed to be the peak k value for many of our prior tests, and we wanted to show the trend toward better performance as k increased (up to a peak point, after which performance should always monotonically decrease).

Abalone - Prediction Accuracy (0/1 Loss)

Algorithm	k=1	k=4	k=8
KNN	0.204	0.216	0.239
ENN	0	0.184	0.228
CNN	0.196	0.216	0.225
K-Means	0.023	0.056	0.035
K-Medoids	N/A	N/A	N/A

Car - Prediction Accuracy (0/1 Loss)

Algorithm	k=1	k=4	k=8
KNN	0.880	0.905	0.917
ENN	0	0.895	0.909
CNN	0.870	0.902	0.911
K-Means	0.725	0.700	0.700
K-Medoids	N/A	N/A	N/A

Segmentation - Prediction Accuracy (0/1 Loss)

Algorithm	k=1	k=4	k=8
KNN	0.829	0.867	0.810
ENN	0	0.838	0.843
CNN	0.862	0.838	0.771
K-Means	0.581	0.281	0.152
K-Medoids	N/A	N/A	N/A

Machine - Prediction MSE

Algorithm	k=1	k=4	k=8
KNN	0.020	0.025	0.027
ENN	0.025	0.020	0.025
CNN	0.027	0.027	0.027
K-Means	0.057	0.027	0.027
K-Medoids	0.027	0.027	0.027

Forest Fires - Prediction MSE

Algorithm	k=1	k=4	k=8
KNN	0.011	0.005	0.004
ENN	0.004	0.005	0.004
CNN	0.005	0.004	0.004
K-Means	0.011	0.006	0.005
K-Medoids	0.004	0.004	0.004

Wine - Prediction MSE

Algorithm	k=1	k=4	k=8
KNN	0.203	0.194	0.195
ENN	0.227	0.194	0.192
CNN	0.203	0.196	0.196
K-Means	0.236	0.237	0.242
K-Medoids	0.242	0.242	0.242

7. Discussion

7.1 Analysis

1. **KNN -** For our regular KNN algorithm, the most accurate results for the classification data sets were obtained with the Car and Segmentation data, while the most accurate results in the regression sets were obtained with the Machine and Forest Fires data. In the regression sets, the discrepancy in the Mean Square Error (MSE) tended not to change much more as the value of k increased. While this was true across the board, it was especially true for the Wine data set where the MSE value is nearly identical across k values 4 and 8, while the same values in Machine and Forest Fires significantly changed. One way to read this, is that it might be due to the fact the classes in the wine data set are clustered closer together. Therefore, the optimal value for k is lower.

   In our hypothesis we predicted that KNN would perform the second best out of all of the algorithms, with the optimum k found somewhere in the middle. Surprisingly, the hypothesis checks out for about half of the data sets - Machine, Wine, and Segmentation all reached peak performance when k equaled 4. This supports the hypothesis that the data was relatively well clustered as taking larger values of k reduced performance. If we were to do this experiment again, one way we could make these results stronger is by taking the entire set 1-8 as values for K to give us a better picture of how performance increases and plateaus/drops with single increments of k.

2. **ENN -** This algorithm performed relatively well. It came up just short of KNN results in early k values, but surpassed KNN as k grew. This makes sense, as we throw out the wrong values, so higher k values should have higher prediction accuracy. This also aligns well with our predictions. While we may have mixed up the ranking for the small k values we did choose, we expected ENN to maintain its first place status as k grows large. Note - there was a bug in our code that messed up the 1-ENN classification values to always return a wrong class. As we chose many values of k, though, we thought it best to continue as normal.

3. **CNN -** This algorithm also performed well. It was a close contender with the other top algorithms. CNN also looked promising for performance increases with increasing k-values on all the data sets except Segmentation. This is what made us rank it 3rd. It is basically tied with other algorithms, but Segmentation was so bad we had to drop it to 3rd among the top 3.

4. **K Means Clustering KNN -** The results were mixed for k-means clustering. It performed very well on the Car data set, moderately well on the Segmentation data set, and very poorly on the rest.

5. **K Medoids Clustering KNN -** The results for k-medoids is a mixed bag. While our mean squared error is low, we also received the same value for each value of k. This should not be possible if our code was working as designed. Also note that for each categorical class, we received an accuracy of 0%. This indicates that we classified no point correctly. This all means that we had a bug in this algorithm that could

not be solved. We were receiving no predictions from the knn_predict() method and therefore our results are inconclusive. With a prediction of 0 for every classification, most of our regression data looks as though it was classified somewhat correctly but that is simply because we normalized all values to be between 0 and 1. We believe our mean squared error values are simply an average of the normalized class values. This would explain why they are the same number because each k value iterates on the same data points.

8. Conclusion

This project involved a significant amount of programming, along with a fair amount of reasoning about the data sets and how best to approach and analyze each one. Decisions were made to normalize all values within the data set so that one feature will not dominate an average. Our base k-nearest neighbor class performed the best. We suspect that using the entire data set is the most prudent because it contains all points that are found in a cluster so outlying points can be classified more readily.

However, with this in mind it is very computationally expensive to use a basic k-nearest neighbor algorithm. The idea of training set reduction decreases computational time for classifying data points by simply decreasing the size of the training set in a way that maintains the information contained within the training data. Looking at our results for edited and condensed nearest neighbor, our accuracy values are quite high and mean squared errors are relatively low. Because these methods reduce the size of the training set, they will perform more quickly when used on larger data sets. Therefore, we recommend to use base k-nearest neighbor for smaller data sets and to choose either edited or condensed nearest neighbor for larger sets.

Acknowledgments

We acknowledge the UCI machine learning repository data sets used for this project (UCI (2019)). A link to the UCI machine learning repository is referenced below.

References

Wes McKinney. Data structures for statistical computing in python, 2010. URL `https://pandas.pydata.org/`.

UCI. UCI machine learning repository, 2019. URL `https://archive.ics.uci.edu/ml/datasets/`.