CSCI 447: K-Nearest Neighbor Analysis

Shane Costello shanecostello@student.montana.edu
Gianforte School of Computing
Montana State University
Bozeman, MT, USA

Hayden Perusich haydenperusich@student.montana.edu
Gianforte School of Computing
Montana State University
Bozeman, MT, USA

Abstract

Our paper investigates K-Nearest Neighbors, Edited K -K-Nearest Neighbors, and K-Means Clustering and their performance on classification and regression. Our k-NN model uses the RBF kernel to predict the target values for regression and a votes system for classification where the closest k neighbors act as the input data for both. For Edited k-NN, we edit the data down to at least 20% of the original data where a point is edited out if it's classified incorrectly. We perform K-Means clustering to derive a reduced data set to be used in the k-NN algorithm. To test each dataset we use 10-fold cross-validation to test each subset. For Classification problems, we use a Confusion Matrix to get the results of the test. This matrix is extrapolated to find Accuracy, Precision, Recall, and F1 scores. For regression, we use Mean Squared Error to determine performance. Through our testing, we found that classification performed better than regression. We also found that the more distributed the classes/target variables were the better the models performed. We believe this is because having a more even distribution allows for k-NN to have a higher chance that when classifying a test point will be surrounded by data points of the correct class/target value.

Key Words: K-Nearest Neighbors, Edited K -Nearest Neighbors, and K-Means Clustering, 10-Fold Cross-cross Validation, Confusion Matrix, Mean Squared Error

1 Introduction

This paper explores the functionality and effectiveness of the k-nearest neighbor (k-NN) algorithm. The algorithm was evaluated on six distinct data sets, three for classification and three for regression. Each data set was divided into three unique subsets to facilitate multiple experiments on the algorithm. The first subset includes the full data set. The second subset was generated using an editing algorithm designed to remove specific data points based on predefined behaviors. The third subset was produced through k-means clustering, where the resulting clusters served as a reduced representation of the data. The purpose of this study is to evaluate how the k-NN algorithm responds to each different data set and each respective subset, for both regression and classification problems. We hypothesize that the subset derived from the editing algorithm will demonstrate the best performance. This process is expected to reduce noise, enhancing the algorithm's predictions. Additionally, we hypothesize that larger data sets will outperform smaller ones. More data will provide a richer information base for training, leading to more accurate predictions. Lastly, we hypothesize that classification datasets will perform better than regression data sets due to the finite nature of classification outputs. The infinite range of potential outputs in regression is expected to create more errors.

2 Experimental Design

This section outlines our experimental design. We describe data preprocessing, including handling missing values, one-hot encoding, and normalization. We then explain the use of ten-fold cross-validation and the experiment's three primary algorithms: k-NN, edited k-NN, and k-means clustering. Finally, we cover the tuning process and evaluation metrics used.

2.1 Preprocess Data

Data sets retrieved from the UCI Machine Learning repository required us to take preprocessing measurements before they could be used for the experiment. We had to handle cases of missing data, transform categorical features into numeric values, and normalize the data sets.

2.1.1 Data Imputation

Cases of absent data were denoted by question marks (?) in all data sets. For the data imputation process, we calculated the mean value of the feature vector where there was missing data. We then input this value in place of any absent data within said feature. We chose to use the mean because it is well fit to the data and will not create an outlier in later calculations.

2.1.2 One-Hot Encoding

The k-NN algorithm requires numeric feature vectors to perform distance calculations. To handle categorical features, we used one-hot encoding. This converts categorical values into binary vectors. For each possible category, a new binary feature vector is created. The feature corresponding to the category present in a given data point is set to 1, while all others are set to 0. This process enables categorical data to be represented numerically, allowing them to be used in distance functions, while not creating a hierarchical relation between the categories.

2.1.3 Data Normalization

To prevent the possibility of feature dominance, we employed data normalization. Feature dominance occurs when features with larger value ranges have a disproportionate influence on distance calculations. Normalization scales all features to exist in the same range so that one feature does not have more influence than another. Considering the previous creation of binary vectors in one-hot encoding, we chose to use 0-1 normalization, scaling all values to be between 0 and 1. This allowed the binary vectors to be already normalized. For each value in a feature vector, the following formula performed this normalization:

$$Normalized\ Value\ = \frac{Value - Feature\ Vector\ Minimum}{Feature\ Vector\ Maximum - Minimum}$$

2.2 Ten-Fold Cross-Validation

We used ten-fold cross-validation in the experiments to reduce the potential of bias in any given division of the data sets. To ensure each fold was representative of the entire data set, we created stratified folds. Before creating folds, for all data sets, an initial 10% of data was removed at random and reserved for the tuning process. This will be discussed in a later section of the paper. The stratification process varied from regression data sets to classification, as discussed in Sections 2.2.1 and 2.2.2.

For all data sets, we allowed for a 20% variance in the size of the folds. When creating stratified folds, it is hard to also mandate folds be exactly equal in size. To put some bounds on this issue of size, we compared the size of the largest fold to the size of the smallest after their creation. If the smaller contained less than 80% of the data found in the larger, we reiterated the process of creating folds until this condition is met. We chose a variance of 20% because it was the smallest value that did not result in an infinite loop. Any number smaller never satisfied the conditions detailed above, hence never accepting a set of folds as adequate.

2.2.1 **Regression Stratification**

Given a preprocessed data set, the creation of stratified folds for regression data sets used the following procedure. First, all the data was sorted in increasing order based on target values. Next, the data was partitioned into groups of ten. Then, we iterate over the set of folds. For a given x^{th} fold, we would iterate over the set of groups of ten, take the data point at index 'x', and put it into the x^{th} fold. This created an even spread of target values throughout the folds. Once this was complete, the values inside each fold and the set of folds were shuffled. This helped to remove any potential ordering of values created in the stratification process.

2.2.2 **Classification Stratification**

Given a preprocessed data set, the creation of stratified folds for classification data sets used the following procedure. First, we counted the number of instances of each class in the data set. This number was divided by the total number of data points to derive the class's prevalence in the data set as a percentage. Each of the folds is to contain that percent of each class. For example, if a data set has 40 instances of Class A, 20 of Class B, and 20 of Class C then each fold will be composed of 40% Class A, 20% Class B, and 20% Class C. Once the folds were populated, the values inside each fold and the set of folds were shuffled. This helped to remove any potential ordering of values created in the stratification process.

2.3 K-Nearest Neighbor

The k-NN algorithm is simple yet effective. It can be used for both, regression and classification problems, with little modification. Given a test data point, the algorithm will calculate the distance between that point and each data point in your training data set. It will then return a list of the k closest data points. In this case, k is some integer value. In the experiment, k is a hyperparameter requiring tuning, as discussed in section 2.7.1. In the experiment, we used the Euclidean distance function to calculate distances between neighbors. We chose to use Euclidean distance because many resources suggested that this is the most common industry practice.

2.3.1 **Regression**

Given a list of a test data point's k-nearest neighbors, we predict the target value by applying a Gaussian kernel. The bandwidth of the kernel, σ, is a hyperparameter that requires tuning, as discussed in section 2.7.2. Once the radial basis function returns a predicted target value, we compare it to the actual target value to verify correctness. If the predicted value is within some error threshold, ε, of the actual value, it is accepted as correct. Epsilon is also a hyperparameter, as discussed in section 2.7.3.

2.3.2 **Classification**

Given a list of a test data point's k-nearest neighbors, we predict a class by employing a plurality vote. That means whatever class is most represented in the set of neighbors becomes the predicted value. Comparison between the predicted and actual class returns a value of either true, the classes match, or false, the classes do not match.

2.4 Edited K-Nearest Neighbor

The edited k-nearest neighbor algorithm is used to construct a smaller subset of a given data set to be used on the k-NN algorithm. To edit the data, we perform the following procedure. We randomly remove one data point from a given data set. This will act as the test set. The remainder of the data set is the training set. We perform k-NN on the singular test point. Depending on whether it is a regression or classification data set, we evaluate the correctness of the prediction accordingly. If the predicted value is correct, the data point remains in the set. If the predicted value is not correct, we remove it from the set. We iteratively perform this process until, either, all data points have acted as the singular test set or the edited data is 20% of its original size. We chose to remove incorrectly predicted data points to try and reduce noise in the edited data set. We stopped the editing process if we reached a reduced size of 20% because we found that certain data sets would result in such small edited sets that they were hard to use in the later k-NN algorithm.

2.5 K-Means Clustering

The k-means clustering algorithm is used to produce a reduced data set to be tested on the k-NN algorithm. *K*, in this algorithm, refers to the number of clusters to be created. In the experiment, we set the number of clusters to be equal to the size of the data set returned from edited k-NN. We chose to do this because we felt the size of the data set from edited k-NN was representative of a properly reduced data set. This also made the comparison between the two reduction methods more accurate, as varying size was not a factor to consider.

Given a training data set, the k-means algorithm works by doing the following. First, it randomly generates *k* centroid locations. It does this by taking the minimum and maximum values from each feature vector and randomly selecting a number in that range. It repeats this process *k* times to generate the appropriate number of centroid locations. Next, we iterate over every data point in the training data set. For each, we use the Euclidean distance function to determine which centroid it is closest to. We assign that data point to the respective centroid. Once all data points have been assigned, we recalculate the centroid locations. For each centroid, we generate the mean of each feature value based on its assigned data points. These new values serve as the new centroid locations. We repeat the process of assigning data points to centroids and regenerating centroid locations until the centroids stop moving.

Once we have the centroid locations, we must assign the centroid values. In the case of classification data sets, this means assigning the centroid a class. In regression data sets, this means generating a target value. This is done by running the k-NN algorithm with the training set and the centroid locations as the test set. We accept the predicted values from k-NN to act as the value for the centroids. The centroid set now represents a reduced data set and can be used for the k-NN algorithm.

2.6 Loss Functions

Loss functions were used to evaluate the performance of the k-NN algorithm; both in the final presentation of results and during the tuning process. Regression and classification data sets required the implementation of two different loss functions.

2.6.1 **Mean Squared Error**

Mean squared error is a measure of how inaccurate the predicted target values were in regression data sets. Error was considered any absolute difference in actual and predicted value greater than the error threshold, epsilon. If the difference was less than epsilon, it was not considered in the error calculation.

$$MSE = \{\frac{1}{n}\sum_{i=1}^{n}(\left|actual_i - predicted_i\right|)^2, \; if \; |actual - predicted| > \varepsilon$$

In this formula, *n* is the total number of data points.

2.6.2 **Confusion Matrix**

A confusion matrix is generated by plotting the predicted classes against the actual classes on a grid plot. From a confusion matrix, we can derive numerous performance metrics by collecting the true positive, false positive, true negative, and false negative values for each class. This facilitates the calculation of both, macro and micro, accuracy, precision, and recall scores. However, the main metric we considered in the experiment was the macro-F1 score. An F1 score represents the harmonic means between precision and recall. We chose to look at a macro score because it conveys how the data set did overall, not on a class-by-class basis. We chose to use the F1 score because we found it to be the most all-encompassing metric, considering it is a combination of precision and recall.

2.7 Hyperparameter Tuning

Hyperparameters are variables used in specific algorithms whose value can be adjusted for optimization. The process of tuning a hyperparameter is finding its optimal value by conducting experiments and comparing performance across various potential values. Tuning was the first step in the experiment following the creation of folds. Before segmenting the data into folds, we reserved a random 10% of the data to be used as the tuning set. In the tuning process, we used 9/10 folds as our training set and the tuning set acts as the test set. We perform cross-validation, so each fold is left out of the training set exactly once. We run the k-NN algorithm with the current training set, trying to predict the values of the tuning set. We aggregate our results across the folds by implementing the respective loss function. Next, we increase the value of the hyperparameter by a preset step size and repeat the process. We repeat the process until we see a decrease in our performance metric when compared to the previous value. For regression, this would be a higher mean squared error. For classification, this would be a lower macro-F1 score. Once we observe a decremented performance, we know we have found an optimal hyperparameter value.

2.7.1 **K**

K is the pinnacle hyperparameter in the experiments. *K* refers to the number of neighbors we examine in the k-NN algorithm. To tune *K*, we started at a value of 1 for all data sets. We chose to start at 1 because it is the smallest possible *K* value. Similarly, *K* has a step size of 1. We chose this to ensure that *K* is always an integer. Moreover, we felt we would find the most optimal value by using the smallest possible step size.

2.7.2 **Sigma (σ)**

Sigma refers to the bandwidth of our Gaussian kernel used to generate predicted target values. We start Sigma at a value equal to 10% of the average range of all feature vectors. Since all the feature vectors are 0-1 normalized in this experiment, our starting Sigma value is 0.1 for all data sets. We chose this value because it is representative of the feature values that will be used in the radial basis function calculation. The Sigma step size was set to 0.01, 1% of the average range of all feature vectors. This value was selected because it will increase Sigma very slowly, aiding in the optimal value selection.

2.7.3 **Epsilon (ϵ)**

Epsilon is the error threshold used in our mean squared error calculation. Since the range of error is dependent on the predicted and actual values, we derived our starting Epsilon value as a percentage of the target value range. Epsilon is started at 1% of the range of target values in a given data set. Epsilon must be scaled to fit each data set because the range of target values varies greatly. Epsilon's step size is equal to 0.1% of the range of target values. The value is intentionally small, so Epsilon is increased slowly.

2.8 The Full Experiment

Our experiment followed a systematic procedure. First, we identified whether the dataset was intended for regression or classification and preprocessed the data accordingly. We set aside the tuning set and created 10 stratified folds for cross-validation. The tuning process began by optimizing K, followed by Sigma, and finally Epsilon using the k-NN algorithm. With the hyperparameters tuned, we performed 10-fold cross-validation on the k-NN algorithm, allowing each fold to serve as the test set exactly once. After completing the cross-validation, we aggregated the results. Next, we initiated another round of 10-fold cross-validation, where the training set was reduced using an editing algorithm. We then applied the k-NN algorithm to the reduced training set and the unedited test set. After completing all iterations, we aggregated the results once more. In the final stage, we performed a third round of 10-fold cross-validation, reducing the training set using the k-means clustering algorithm. Once clustered, we applied the k-NN algorithm to the reduced training set and the original test set. After all iterations were completed, the results were again aggregated. This entire process was repeated for each of the six datasets.

3 Software Design

We employed an object-oriented approach to software design, modeling each major component of our experiment using a class. We had a class to store all of our data, including the 10 folds and the tuning set. Another class was used to model hyperparameters, with each of the three hyperparameters being instances of this class. Additionally, there was a class to represent each of the following algorithms: k-NN, editing, and k-means clustering. Finally, our loss functions were encapsulated in two classes: one for mean squared error and one for the confusion matrix. All programming was done in Python.

4 Results

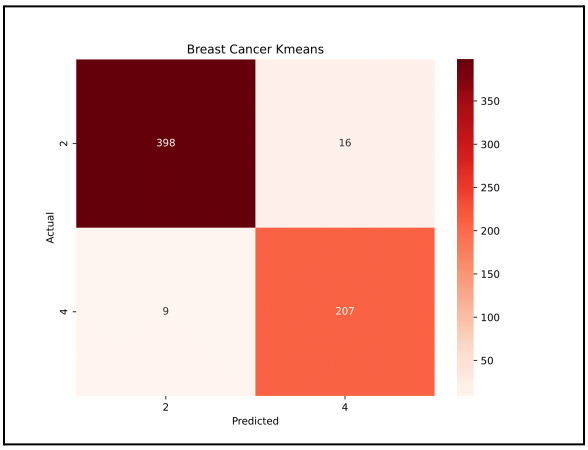

Figure 1:
Breast Cancer k-NN Confusion Matrix

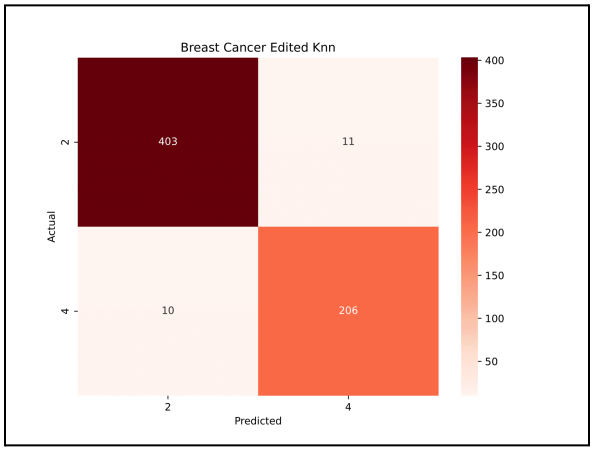

Figure 2:
Breast Cancer Edited k-NN Confusion Matrix

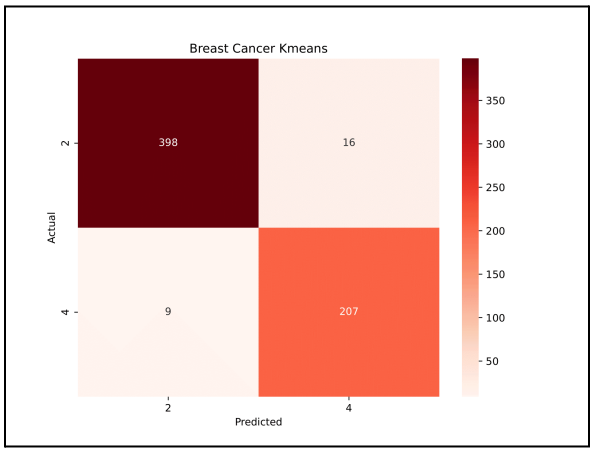

Figure 3:
Breast-Cancer K-Means k-NN Confusion Matrix

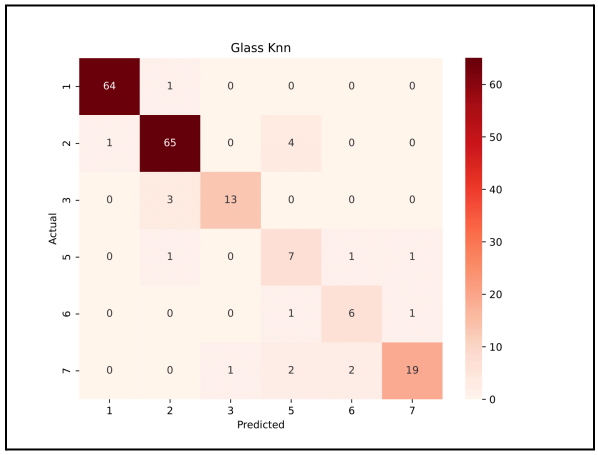

Figure 4:
Glass k-NN Confusion Matrix

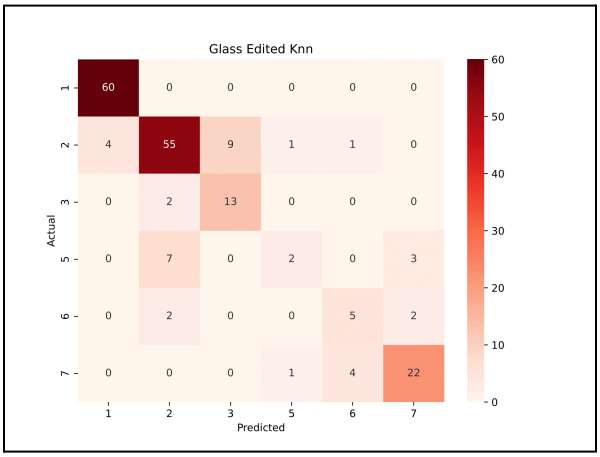

Figure 5:
Glass Edited k-NN Confusion Matrix

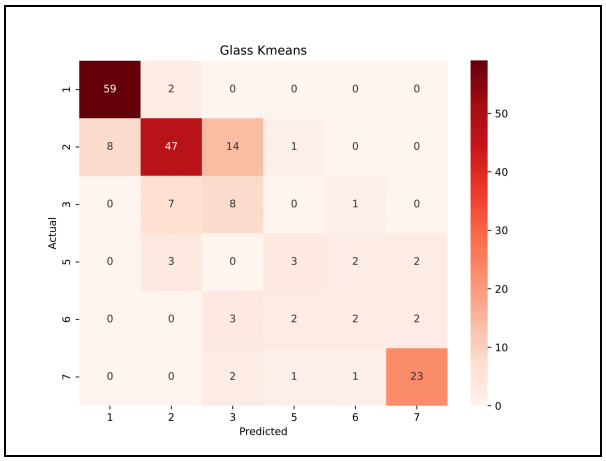

Figure 6:
Glass K-Means k-NN Confusion Matrix

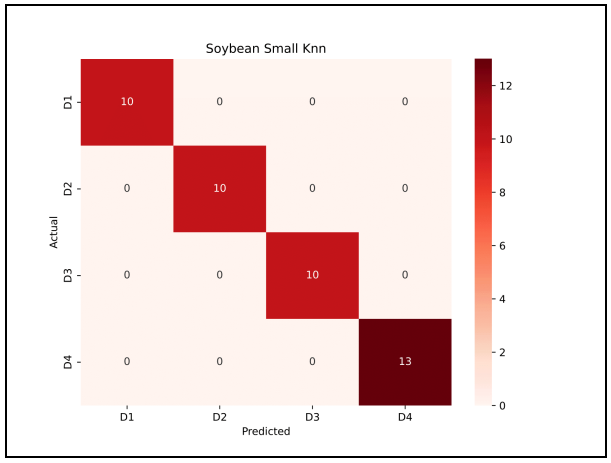

Figure 7:
Soybean k-NN Confusion Matrix

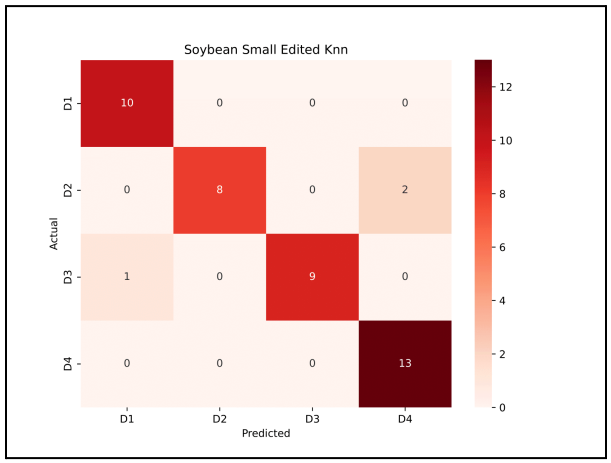

Figure 8:
Soybean Edited k-NN Confusion Matrix

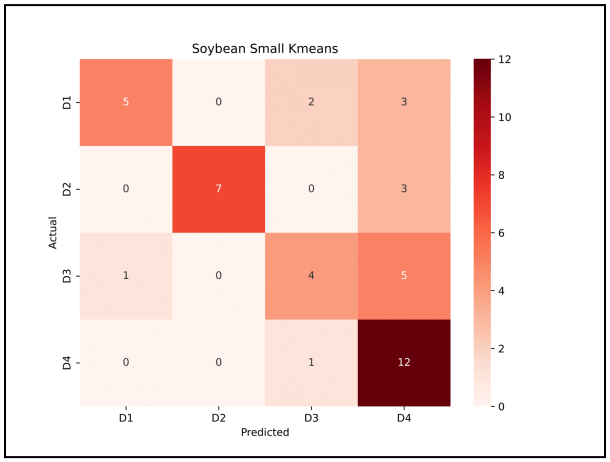

Figure 9:
Soybean K-Means k-NN Confusion Matrix

Figure 10:
Regression Data Sets Mean Squared Error

Forest Fires		
K_means	*Edited_Knn*	*Knn*
1841.294	1347.161	3873.556
Machine		
K_means	*Edited_Knn*	*Knn*
26233.234	11552.579	5778.645
Abalone		
K_means	*Edited_Knn*	*Knn*
10.533	8.521	6.475

5 Discussion

For classification, we found that KNN did exceptionally better than the other 2 models in terms of F1 score, and overall accuracy. We believe this is because without adding noise like the last project all the data points, especially if the target class is evenly distributed among the data points, help our KNN model correctly predict that class rather than throw it off. When we cut down on the data whether it's through editing the data set or K Means Clustering we remove data points that help reinforce our models' correct predictions. Also, removing the data in edited KNN if a certain class appears less than other classes (3 and 4 in Fig 5) reduces our KNN model's ability for correct classification and it guesses a class that appears more in the data set. (ex. Class 1 and 3 in Fig 5).

The one data set that acts as an exception to this rule is the forest fires dataset. On this dataset, both Edited KNN and K Means KNN performed better than regular KNN. We believe this is because of the distribution of target values in the dataset. In forest fires the vast majority of target values equal 0. This creates an uneven distribution that causes the KNN model to incorrectly predict 0 for many of the data points. The reason we believe both of the other models perform better than KNN is because removing many of these target value 0 data points helps to create a more even distribution of data allowing those false 0 to be more accurately predicted. Fig 10.

Much like our previous project (Naive Bayes Classification) KNN also struggles more when it has more classes to choose from. The glass dataset is a prime example where when you have 6 different classes the number that are incorrectly classified is much greater than the other datasets. For regular KNN our macro F1 score is 0.82 whereas for K_Means classification it's 0.58. This means that by applying K Means clustering we are losing data points that are useful for classification. That is why our performance degrades so much when applying either Edited KNN or K Means Classification. Especially with glass that only has 214 data points in the whole data set which for 6 classes isn’t a whole lot. Fig 5.

The classification dataset that performed the best by far was the soybean-small set. With the KNN model, it got a f1 score of 1 and with EditedKNN it got an f1 of 0.92. This performance we believe can be attributed to the even distribution of the classes in the data set and the relatively small number of classes. This allowed our models to accurately predict the classes without the same problems datasets like glass had. A point of note for the soy-bean dataset is it only contains 47 data points which reduces our ability to test its performance as the number of test points is much less than other datasets. This would affect our results by overtraining on such a small dataset reducing its ability to generalize on data points not in the set. Fig 7.

For both classification and regression, our k Mean KNN model performed the worst over the datasets. We attribute this to our clusters having a weak correlation to our classes and target values indicating possible non-separability between the classes and target values for our feature set. Fig 3, 10.

6 Conclusion

We conclude that we were only partially right with our hypothesis. The subsets given by editing the data in edited KNN only gave us significantly better results for the forest fire data set than the other models. For all other datasets, our results were similar or worse compared to KNN. We believe this is because when editing our data down, while it did improve performance we removed certain data points that proved to be important when predicting. This is shown in our reduction of performance along many of our datasets where their rate of false positives was greater with edited KNN than KNN. It's important to note that the dataset that performed the best with edited KNN, forest fires, had many target values equal to 0. Removing these data points helped our performance by reducing the bias to 0 that the KNN model had.

We found that the number of classes for classification and the distribution of target values for regression had a much larger effect on our model performance than the number of data points in each dataset. We believe this is because, with an even distribution of target values/classes, our models were better able to predict accurately no matter the size of the data set itself. The SoyBean dataset is a prime example of this where while it was by far the smallest dataset we tested it provided the highest f1 score among the classification datasets.

In terms of our last hypothesis, our models did prove to be more accurate with classification as opposed to regression. As hypothesized we believe this is because when classifying finite values our models were able to be more determinate. This is contrasted with regression where the target values have an infinite possible number of values and hence more difficult for our models to predict accurately.

Overall we found that classic KNN was able to predict the most accurate amount of our set of models. This is because when given a dataset with fairly well-distributed classes/target values having a high number of data points allowed KNN to reinforce our correct prediction. Edited KNN did the second best among the models. We believe this is because through editing the data we can retain the targeted class value, unlike K Means KNN where that value is edited. We found that the more levels of editing and clustering we applied to our preprocessed data the worse the models performed.

Appendix A

In this appendix, we will explore the division of labor and contributions from each team member.

Paper

Abstract: Hayden Perusich
Section 1: Shane Costello
Section 2: Shane Costello
Section 3: Shane Costello
Section 4: Hayden Perusich
Section 5: Hayden Perusich
Section 6: Hayden Perusich

Video

Script: Hayden Perusich
Recording: Shane Costello
Edit: Shane Costello

Code

confusion_matrix.py: Shane Costello
data.py: Shane Costello
demo.py: Hayden Perusich
edited_knn.py: Hayden Perusich
hyperparameter.py: Shane Costello
k_means.py: Shane Costello
knn.py: Hayden Perusich
loss.py: Shane Costello
tune.py: Shane Costello