k-Nearest Neighbor and Decision Trees

COSC528 - Fall 2018
Project 2
11/05/18

Rekesh Ali

1 Introduction

This goal of this project is to use k-nearest neighbor (kNN) and decision trees (DT) to classify breast cancer patients as either benign or malignant based on a number of symptoms they display. There are many factors that go into each algorithm, such as neighbor limit for kNN or maximum impurity for decision trees, so there will be a lot of hyper-parameter optimization. We will be using many different performance metrics to analyze the fidelity of our model and optimize our hyper-parameters for a most confident solution. In addition, PCA will be used to reduce the dimensionality of the dataset for decision tree estimation.

2 Data Preparation

The data acquired describes 699 instances split into 11 classes with unique, discrete, and numeric data types. The first class is an ID so that will be discarded. The last class is strictly discrete and can take on a label of 2 or 4, meaning benign and malignant respectively. The features in between these classes are reported in intervals of 1 on a scale of 1 to 10, and they represent the level of the attribute displayed by the patient, such as 'Clump Thickness' or 'Bare Nuclei' for example. Because the values for these features are tractable in variety and because they represent the degree to which an instance resides along a dimension, we have the unique ability to treat these features as discrete or numeric without adding any computational stress. However, PCA of the dataset will introduce too many different numbers and make a discrete algorithm unfeasible, so this assumption is only valid with the original dataset. Lastly, there are 17 instances with unmarked values. A reasonably quick and simple way to impute these values is to use the kNN or DT to train the data for that attribute and solve for that instance. Another way is to work backwards in a DT from a leaf with the same output class until getting to a node whose index matches that of the missing attribute, and getting the split value. This study has plenty of data to work with, however, so these instances will be ignored.

3 Implementation

3.1 Model

3.1.1 k-Nearest Neighbor

kNN is rooted in statistical certainties and is a very simple algorithm. The logic is quite clear: for a given instance $\mathbf{x}_t$ in a test set $\mathbf{X} = [\mathbf{x}_1 ... \mathbf{x}_t]^T$, its distance to every instance $\mathbf{y}_m$ in a training set $\mathbf{Y} = [\mathbf{y}_1 ... \mathbf{y}_m]^T$ is computed and stored. When all instances in the training set have been exhausted, the distances are sorted in ascending order where the first k samples are the k closest to the test sample. These instances now reside in the 'bin' describing

the test sample's relationship to the training set, or in other words all members of the bin represent a local attribute dominance of the training set, which by definition extends to the attribute containing the class. Thus, if the test sample was also in the training set, it would reside in that bin as well and have the same likelihood of belonging to the dominant class in that locale. With that in mind, the probability p^i that $\mathbf{x}_t$ belongs to class C^i is simply the number of neighbors n in class C^i divided by the total number of neighbors k, where k_n^i is 1 if the n-th neighbor belongs to class C^i and 0 otherwise. The class distribution of these k-neighbors can be computed with **Eq.1**, and the class assignment is simply the class with the highest probability. In the case of a tie, our algorithm will use the class that makes the first appearance in the set of k-neighbors. The distances will be computed according to **Eq.2**, where L indicates L-norm level. The hyperparameters associated with this method are k, bin size, and L, norm type.

$$p^i = \frac{\sum_{n=1}^{k} k_n^i}{k} \tag{1}$$

$$C_t = C^i \text{ :: } \max(p^i)$$

$$d_m = \sqrt[L]{|\mathbf{x}_t - \mathbf{y}_m|^L} \tag{2}$$

3.1.2 Decision Tree

Decision trees are structures that 'ask' questions of a test sample, where at each step the answer leads the sample down further to some ultimate destination, aka its classification. The tree is constructed of nodes that ask these questions, terminals or leaves that classify the instance, and paths or branches that connect nodes to nodes or nodes to terminals. A very simple idea in theory, but not so straightforward in implementation. For instance, how does the user know what question to have the tree ask at each node? For any given dataset, there is an immense number of combinations leading to an unquantifiable number of trees, each with their own strengths and weaknesses, and little guarantee of being useful. Fortunately, one does not actually need to know any of the questions in order to generate a tree, they must simply leverage a little bit of information theory. Namely, that the impurity of a list of values depends on the variation of the values within that list. That is, if most values are the same, impurity is low. In other words, low impurity means high likelihood of a value. **Eqs.3-5** are a few standard measures for the impurity, ϕ, of a binary class system where p is the proportion of values (from **Eq.1**) that belong to the first class of the binary system.

$$\text{Entropy: } \phi = -plog_2(p) - (1-p)log_2(1-p) \tag{3}$$

$$\text{Gini Index: } \phi = 2p(1-p) \tag{4}$$

$$\text{Misclassification Error: } \phi = 1 - \max(p, 1-p) \tag{5}$$

Let us say for a given training set $\mathbf{Y}$ with N rows and D dimensions describing its attributes, there is a way to split the rows into n more branches along an attribute d such

that the impurity of the split thereafter is the minimum impurity for any possible splitting of the rows along any of the given attributes. The impurity of a split is a bit different from the definition above, because it takes into account all of the resulting nodes. The split impurity is a weighted average of all of the impurities ϕ_j of the resulting n branches, **Eq.6**. The index d is recorded as the split index for that node, meaning a test sample that arrives at that node will be checked at that attribute. This process continues until reaching a maximum depth or until dipping below a maximum impurity, thus resulting in a leaf on the tree. The leaf records the class information for that series of decisions; the value of which is equal to the class with the highest proportion in that leaf. The hyperparameters for this approach are maximum allowable entropy and maximum allowable tree depth.

$$\phi_n = \sum_{j=1}^{n} \frac{N_j}{N_n} \phi_j \tag{6}$$

Discrete and numeric data types are treated a bit differently in DT's. For discrete data, nodes are split into branches containing homogenous values found at that attribute. That means for every node, there are as many splits as there are different values in the rows at the split attribute. For numeric data, this study sorts the values at that attribute in ascending order and takes every possible side by side 2 way split into the impurity minimizer; side by side meaning each split is constrained such that the resulting branches are above and below a value. The barebones algorithm is below, where n is a list of lists containing the indices associated with each branch.

```
GenerateTree(X):
    φ = NodeImpurity(X)
    if φ < φ_max :
        AddLeaf(X)
        return
    d,n = SplitAttribute(X)
    AddNode(d)
    for j in n:
        AddBranch(X[j])
        GenerateTree(X[j])
    return
```

3.1.3 Principal Component Analysis

The PCA is done by performing a singular value decomposition (SVD) on the numerical data matrix. This program uses python's built in svd function in the linear algebra package

of numpy. The decomposition is illustrated below, where $\mathbf{U} = [\mathbf{u}_1 ... \mathbf{u}_t]$ are the left singular column vectors denoting the principal components, Σ is a diagonal matrix containing the singular values which represent the square of the variance, σ_j^2, in each principal direction, and $\mathbf{V}^* = [\mathbf{v}_1 ... \mathbf{v}_j]^T$ are the right singular row vectors. The product $\sigma_1 \mathbf{u}_1$ represents the first principal component weighted by its variance, which is the direction within the dataset that captures the most variance.

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^* \tag{7}$$

The sum of the root of the singular values represents the total variance within the dataset. Luckily, the singular values are ranked by magnitude, so the principal components are in order by variance. Also, it usually does not take too many components to recover a high percentage of the variance, so data can be represented as much lower cost given that there are a low amount of k singular values that capture a high percentage of the variance. Thus the data matrix can thus estimated using the first k principal components, by plugging in the values shown below into **Eq.2** above.

$$\begin{aligned}
\mathbf{X}_k &\approx \mathbf{X} \\
\mathbf{U}_k &= [\mathbf{u}_1 ... \mathbf{u}_k] \\
\Sigma_k &= diag[\sigma_1^2 ... \sigma_k^2] \\
\mathbf{V}_k^* &= [\mathbf{v}_1 ... \mathbf{v}_k]^T
\end{aligned}$$

3.2 Program

The implementation of the models above results in a very computationally heavy program overall. Both the kNN and the DT randomize the dataset before splitting into testing and training. This alone is not a good benchmark for comparison because the instances will be unique for each hyperparameter run, so the models will be iterated 100 times and the results will contribute to a sum over all the iterations for a given hyperparameter combination. Factoring in all the different hyperparameter combinations along with the 100 iteration standard, it is clear why the computation is so heavy. Nevertheless, it will illustrate a real, stochastic process which is good for the analysis. After the classes for the test set are predicted during each iteration, the program finds: # of true positives (TP), # of true negatives (TN), # of false positives (FP), and # of false negatives (FN). These values are added to the previous at each iteration. After all iterations for that hyperparameter combination are complete, the performance metrics can found. The ones we will be looking at are: true positive rate (TPR), positive predicted value (PPV), true negative rate (TNR), and F1 score (F1S). The equations for these metrics are below. Once all the hyperparameter combinations have been looped through, the relevant data is printed to an excel file. main.py contains the iterative hyperparameter testing, explore.py and prepare.py read and edit the

data, implement.py has the kNN, DT, and PCA, and finally output.py records the results. The current build can be found on github at rekeshali/kNN-DecisionTree.

$$
\begin{aligned}
ACC &= (TP + TN)/(TP + FP + TN + FN) \\
TPR &= TP/(TP + FN) \\
PPV &= TP/(TP + FP) \\
TNR &= TN/(TN + FP) \\
F1S &= 2 \times PPV \times TPR/(PPV + TPR)
\end{aligned}
$$

4 Results

4.1 k-Nearest Neighbor

The hyperparameters for kNN are k for neighbors in bin and L for L-norm type. Values of 2 through 8, 17, and 33 were selected for k, and values of 1 through 10 were selected for L. **Table 1** is the confusion matrix for each hyperparameter combination, where the matrices are color coded with a legend in the top left for readability. The yellow and red are cases that were classified correctly, while the orange are cases that were classified incorrectly. Judging from the relatively low orange numbers, the kNN seems to be running according to plan.

TN	FP	k-neighbors																	
FN	TP	2		3		4		5		6		7		8		17		33	
Lnorm	1	10848	229	10880	241	10901	269	10902	272	10922	250	10941	232	10923	224	10947	225	10925	228
		480	5543	234	5745	288	5642	296	5630	340	5588	326	5601	323	5630	384	5544	473	5474
	2	10834	288	10836	261	10866	262	10850	282	10882	238	10774	273	10807	253	10905	247	10856	217
		478	5500	254	5749	260	5712	221	5747	269	5711	270	5783	283	5757	306	5642	421	5606
	3	10784	281	10778	295	10822	270	10926	288	10849	284	10852	267	10831	278	10802	260	10926	231
		566	5469	246	5781	278	5730	207	5679	292	5675	253	5728	277	5714	299	5739	408	5535
	4	10831	321	10757	325	10823	295	10783	327	10806	329	10777	321	10737	298	10869	280	10863	211
		558	5390	266	5752	300	5682	242	5748	254	5711	249	5753	282	5783	270	5681	385	5641
	5	10912	329	10889	343	10858	319	10828	306	10807	302	10807	321	10862	332	10826	297	10979	225
		515	5344	219	5649	289	5634	227	5739	239	5752	231	5741	276	5630	258	5719	378	5518
	6	10789	320	10822	317	10837	333	10844	313	10743	342	10630	349	10815	320	10786	274	10853	241
		556	5435	273	5688	258	5672	226	5717	256	5759	233	5888	234	5731	240	5800	386	5620
	7	10821	303	10750	320	10752	316	10759	342	10920	306	10779	320	10815	328	10728	319	10779	241
		533	5443	227	5803	273	5759	229	5770	265	5609	219	5782	240	5717	242	5811	393	5687
	8	10807	294	10768	309	10867	327	10730	330	10808	335	10853	330	10774	327	10812	279	10893	217
		513	5486	254	5769	308	5598	219	5821	221	5736	194	5723	238	5761	253	5756	383	5607
	9	10836	351	10727	329	10892	315	10807	337	10740	315	10791	303	10810	312	10853	289	10872	246
		520	5393	257	5787	327	5566	232	5724	238	5807	272	5734	244	5734	216	5742	382	5600
	10	10890	323	10740	326	10840	300	10775	324	10864	296	10817	363	10735	332	10801	288	10803	242
		504	5383	259	5775	303	5657	220	5781	288	5652	228	5692	264	5769	268	5743	410	5645

Table 1: kNN confusion matrix for every hyperparameter combination.

Let us take a look at the accuracy for each hyperparameter combination. The accuracy is a measure of how many samples were classified versus how many samples there are, or in other words it is the probability that a case will get classified correctly. **Table 2** shows the accuracy for the kNN algorithm, where the colors from red to green indicate low and high

values, respectively. The values in this table are much more intuitive, and we can finally see where some parameter values may shine over others. For example, columns $k = 2, 33$ both have low relative values, so those are probably not good to use if one is optimizing for accuracy. That said, all of the accuracies reported here are very high in a global sense, so perhaps one should not optimize on something that could potentially net little returns, as opposed to something else that may be slacking hard and have more room to grow. Regardless, a k value of 5 does well for all the L values. As far as L-norm values go, it appears the classic L2 seems to consistently hit the high mark for nearly all k values. The highest accuracy belongs to a k value of 3 with the L1 norm. It appears L-norm does not do too much to the accuracy, because the columns are generally more uniform than the rows.

ACC		k-neighbors								
		2	3	4	5	6	7	8	17	33
Lnorm	1	95.854%	97.222%	96.743%	96.678%	96.550%	96.737%	96.801%	96.439%	95.901%
	2	95.520%	96.988%	96.947%	97.058%	97.035%	96.825%	96.865%	96.766%	96.269%
	3	95.047%	96.836%	96.795%	97.105%	96.632%	96.959%	96.754%	96.731%	96.263%
	4	94.860%	96.544%	96.520%	96.673%	96.591%	96.667%	96.608%	96.784%	96.515%
	5	95.064%	96.713%	96.444%	96.883%	96.836%	96.772%	96.444%	96.754%	96.474%
	6	94.877%	96.550%	96.544%	96.848%	96.503%	96.596%	96.760%	96.994%	96.333%
	7	95.111%	96.801%	96.556%	96.661%	96.661%	96.848%	96.678%	96.719%	96.292%
	8	95.281%	96.708%	96.287%	96.789%	96.749%	96.936%	96.696%	96.889%	96.491%
	9	94.906%	96.573%	96.246%	96.673%	96.766%	96.637%	96.749%	97.047%	96.327%
	10	95.164%	96.579%	96.474%	96.819%	96.585%	96.544%	96.515%	96.749%	96.187%

Table 2: kNN percent accuracy for every hyperparameter combination.

TPR		k-neighbors								
		2	3	4	5	6	7	8	17	33
Lnorm	1	92.031%	96.086%	95.143%	95.005%	94.265%	94.500%	94.574%	93.522%	92.046%
	2	92.004%	95.769%	95.646%	96.297%	95.502%	95.539%	95.315%	94.855%	93.015%
	3	90.621%	95.918%	95.373%	96.483%	95.106%	95.770%	95.376%	95.048%	93.135%
	4	90.619%	95.580%	94.985%	95.960%	95.742%	95.851%	95.350%	95.463%	93.611%
	5	91.210%	96.268%	95.121%	96.195%	96.011%	96.132%	95.327%	95.683%	93.589%
	6	90.719%	95.420%	95.649%	96.197%	95.744%	96.193%	96.077%	96.026%	93.573%
	7	91.081%	96.235%	95.474%	96.183%	95.489%	96.351%	95.971%	96.002%	93.536%
	8	91.449%	95.783%	94.785%	96.374%	96.290%	96.721%	96.033%	95.790%	93.606%
	9	91.206%	95.748%	94.451%	96.105%	96.063%	95.471%	95.918%	96.375%	93.614%
	10	91.439%	95.708%	94.916%	96.334%	95.152%	96.149%	95.624%	95.542%	93.229%

Table 3: kNN percent true positive rate for every hyperparameter combination.

Table 3 shows the TPR for kNN, which is the probability that a case will get classified positive when it should be. It looks very much like the previous table, which is quite obvious

since the equation for TPR is a subset of the equation for ACC. That said, the low valued trends are mostly identical, but the higher valued TPRs have shifted away from the L2 norm and down into higher ones. A k of 5 is still consistently good here, which is a subtle reminder that disposes us to tune our model in that direction. This time, we can see a rather large disparity between the min and max values here, so this table is a better indicator than the ACC one of what hyperparameters to choose for optimization.

PPV		k-neighbors								
		2	3	4	5	6	7	8	17	33
Lnorm	1	96.033%	95.974%	95.449%	95.391%	95.718%	96.023%	96.174%	96.100%	96.001%
	2	95.024%	95.657%	95.614%	95.323%	95.999%	95.492%	95.790%	95.806%	96.273%
	3	95.113%	95.145%	95.500%	95.173%	95.234%	95.546%	95.360%	95.666%	95.994%
	4	94.379%	94.652%	95.064%	94.617%	94.553%	94.715%	95.099%	95.303%	96.394%
	5	94.201%	94.276%	94.641%	94.938%	95.012%	94.705%	94.431%	95.063%	96.082%
	6	94.440%	94.721%	94.455%	94.809%	94.394%	94.404%	94.712%	95.489%	95.888%
	7	94.727%	94.774%	94.798%	94.404%	94.827%	94.756%	94.574%	94.796%	95.935%
	8	94.913%	94.916%	94.481%	94.635%	94.482%	94.548%	94.629%	95.377%	96.274%
	9	93.889%	94.621%	94.644%	94.440%	94.855%	94.981%	94.840%	95.208%	95.792%
	10	94.339%	94.657%	94.964%	94.693%	95.024%	94.005%	94.558%	95.225%	95.889%

Table 4: kNN percent positive predicted value for every hyperparameter combination.

TNR		k-neighbors								
		2	3	4	5	6	7	8	17	33
Lnorm	1	97.933%	97.833%	97.592%	97.566%	97.762%	97.924%	97.990%	97.986%	97.956%
	2	97.411%	97.648%	97.646%	97.467%	97.860%	97.529%	97.712%	97.785%	98.040%
	3	97.460%	97.336%	97.566%	97.432%	97.449%	97.599%	97.498%	97.650%	97.930%
	4	97.122%	97.067%	97.347%	97.057%	97.045%	97.108%	97.300%	97.489%	98.095%
	5	97.073%	96.946%	97.146%	97.252%	97.281%	97.115%	97.034%	97.330%	97.992%
	6	97.119%	97.154%	97.019%	97.195%	96.915%	96.821%	97.126%	97.523%	97.828%
	7	97.276%	97.109%	97.145%	96.919%	97.274%	97.117%	97.056%	97.112%	97.813%
	8	97.352%	97.210%	97.079%	97.016%	96.994%	97.049%	97.054%	97.484%	98.047%
	9	96.862%	97.024%	97.189%	96.976%	97.151%	97.269%	97.195%	97.406%	97.787%
	10	97.119%	97.054%	97.307%	97.081%	97.348%	96.753%	97.000%	97.403%	97.809%

Table 5: kNN percent true negative rate for every hyperparameter combination.

Table 4 is the PPV which can be interpreted as the probability that a case is classified positive correctly, which for a cancer patient is a measure of how much they should believe their malignant diagnosis. Again, the colors may mask the truth here, which is that all of these numbers are very close to each other. That said, it appears the higher L-norm values are really taking a hit here, which is much like the ACC table. Also, the k situation seems to have reversed, where now the more moderate values are lower and the higher values that

were not so good in the other tables are consistently on top. However, it still appear a k of 2 is not ideal by any metric. **Table 5** is the true negative rate, a.k.a. the probability that a negative classification is true. This table looks like the previous one, but a bit more exaggerated. The L1 and L2 norms seem to be doing the best, along with the larger bin sizes. There appears to be an obvious barrier between low and high values, where the yellow separates the greens and reds. Overall, these values are on the higher end relative to the previous tables.

F1S		k-neighbors								
		2	3	4	5	6	7	8	17	33
Lnorm	1	93.989%	96.030%	95.296%	95.198%	94.986%	95.255%	95.367%	94.794%	93.982%
	2	93.490%	95.713%	95.630%	95.807%	95.750%	95.516%	95.552%	95.328%	94.616%
	3	92.813%	95.530%	95.436%	95.824%	95.170%	95.658%	95.368%	95.356%	94.543%
	4	92.461%	95.114%	95.025%	95.284%	95.144%	95.280%	95.225%	95.383%	94.982%
	5	92.681%	95.261%	94.880%	95.562%	95.509%	95.413%	94.877%	95.372%	94.819%
	6	92.542%	95.069%	95.048%	95.498%	95.064%	95.291%	95.389%	95.757%	94.716%
	7	92.868%	95.499%	95.135%	95.285%	95.157%	95.547%	95.267%	95.395%	94.720%
	8	93.149%	95.347%	94.633%	95.497%	95.377%	95.622%	95.326%	95.583%	94.921%
	9	92.528%	95.181%	94.547%	95.265%	95.455%	95.225%	95.376%	95.788%	94.691%
	10	92.866%	95.179%	94.940%	95.506%	95.087%	95.065%	95.088%	95.383%	94.540%

Table 6: kNN percent F1 score for every hyperparameter combination.

Finally, **Table 6** shows the F1 score for each hyper parameter combination. The F1 score is a non-linear combination of the TPR and the PPV, and it appears to have traits from the both of them. That is, the high and low k disparity is back from the TPR, but the values from the higher norms are trending in the PPV direction. That said, the L2 norm is still tough to beat, and k at 5 is just as consistent as ever. Overall these values have been quite high all across the board.

4.2 Decision Tree

The hyperparameters for the decision tree are maximum impurity threshold and maximum tree depth. Since these parameters are conditional and the dataset is continuously randomized, there is no telling when one will activate over the other, so they really cannot be tested against each other similar to how the kNN parameters were. Therefore, we will have to analyze them separately. There are also a lot of other adjusments to the model that we can make, such as datatype and impurity type, so those results are included as well for completion.

Table 7 shows the metrics for the decision tree for varying values of the impurity threshold. First off, a few words about the table layout. The color coding is identical to all the previous tables. Depth Max is not the value set for maximum depth threshold, that is in

		Max Impurity							
		Discrete				Numeric			
		0.1	0.2	0.3	0.4	0.1	0.2	0.3	0.4
Entropy	ACC	94.696%	94.947%	94.901%	95.111%	93.772%	93.357%	93.064%	92.649%
	TPR	91.504%	91.535%	91.831%	92.128%	86.826%	86.075%	84.854%	83.336%
	PPV	93.231%	93.804%	93.344%	93.789%	94.676%	94.560%	94.818%	95.225%
	TNR	96.417%	96.770%	96.528%	96.717%	97.429%	97.311%	97.497%	97.723%
	F1S	92.360%	92.656%	92.581%	92.951%	90.581%	90.118%	89.560%	88.885%
	Depth Max	4	5	5	5	12	12	12	11
	Depth Min	4	4	4	4	6	6	2	2
Gini Index	ACC	95.123%	95.497%	95.281%	95.713%	92.439%	92.450%	92.421%	92.152%
	TPR	92.029%	93.001%	92.615%	94.376%	82.980%	84.201%	84.627%	84.049%
	PPV	93.799%	93.982%	93.937%	93.389%	94.978%	93.690%	93.065%	92.665%
	TNR	96.767%	96.827%	96.736%	96.428%	97.604%	96.924%	96.610%	96.462%
	F1S	92.906%	93.489%	93.271%	93.880%	88.575%	88.692%	88.646%	88.147%
	Depth Max	4	4	4	4	11	9	2	2
	Depth Min	4	4	4	3	4	2	2	2
Misclassification Erro	ACC	95.170%	95.129%	95.269%	65.427%	92.532%	92.339%	92.158%	64.561%
	TPR	92.678%	92.566%	93.338%	0.000%	83.477%	83.561%	83.401%	0.000%
	PPV	93.429%	93.631%	93.103%		94.711%	93.951%	93.337%	
	TNR	96.506%	96.538%	96.302%	100.00%	97.462%	97.089%	96.826%	100.00%
	F1S	93.052%	93.096%	93.220%		88.740%	88.452%	88.090%	
	Depth Max	4	4	4	1	5	2	2	1
	Depth Min	4	3	2	1	2	2	2	1

Table 7: Decision tree metrics controlled by maximum impurity.

fact set to something unattainably high to get rid of interference. Depth Max is actually the maximum depth of a tree encountered during all 100 iterations of the parameter combination. Depth Min is the smallest tree encountered. On the note of tree depth, it appears the discrete type trees have less levels than their numeric counterparts. This is likely due to the fact that each discrete node can have as many branches as there are values in the attribute, thus spreading the tree out and covering more options in a shorter space. It appears for both data types and for all impurity types that as we increase the impurity threshold, there are less levels appearing in the tree. This is due to the impurity meeting that threshold really quickly. For the misclassification error this happens immediately, because if the classes aren't equally frequent, one will have a higher probability than the other and our misclassification error will be less than 0.4 off the bat. In general, however, it looks like impurity threshold doesn't really do much in terms of change here because the colors are continuing horizontally across the table. The bigger source for change here seems to be the data type treatment. That is, it looks like the discrete method is working much better than the numeric one, and is a much faster classifier since there are less levels. The impurity measurements are interesting as well. Gini index and misclassification error are nearly identical, but the entropy minimizer is different and apparently the worst choice here for discrete data (disregard 0.4).

		Max Depth											
		Discrete						Numeric					
		2	3	4	5	7	10	2	3	4	5	7	10
Entropy	ACC	91.942%	92.842%	93.000%	94.602%	94.883%	94.942%	92.392%	93.351%	94.491%	94.772%	94.433%	93.842%
	TPR	87.644%	87.323%	89.423%	91.369%	91.815%	91.263%	87.598%	91.966%	92.317%	92.949%	89.725%	87.200%
	PPV	89.163%	91.832%	90.084%	93.094%	93.369%	94.268%	90.470%	88.970%	91.821%	92.081%	94.247%	94.858%
	TNR	94.258%	95.816%	94.866%	96.346%	96.520%	96.958%	94.993%	94.071%	95.643%	95.743%	97.006%	97.440%
	F1S	88.397%	89.521%	89.753%	92.223%	92.585%	92.741%	89.011%	90.443%	92.068%	92.513%	91.931%	90.868%
	Depth Max	2	3	4	4	4	4	2	3	4	5	7	10
	Depth Min	2	3	4	4	4	4	2	3	4	5	6	6
Gini Index	ACC	91.854%	92.988%	93.550%	95.211%	95.199%	94.906%	92.398%	92.906%	93.374%	93.620%	92.825%	93.415%
	TPR	87.143%	87.913%	89.593%	92.206%	92.287%	91.755%	84.895%	85.350%	86.910%	86.724%	84.095%	85.991%
	PPV	89.225%	91.831%	91.730%	94.122%	93.958%	93.749%	92.548%	93.734%	93.545%	94.772%	94.866%	94.721%
	TNR	94.373%	95.748%	95.671%	96.853%	96.779%	96.638%	96.376%	96.948%	96.811%	97.387%	97.541%	97.417%
	F1S	88.172%	89.830%	90.649%	93.154%	93.115%	92.741%	88.556%	89.346%	90.106%	90.570%	89.156%	90.145%
	Depth Max	2	3	4	4	5	4	2	3	4	5	7	10
	Depth Min	2	3	4	4	4	4	2	3	3	3	3	2
Misclassification Erro	ACC	91.819%	92.637%	93.474%	95.251%	95.117%	95.386%	92.205%	92.275%	92.316%	92.398%	92.287%	92.520%
	TPR	87.342%	87.421%	88.995%	92.443%	91.965%	93.252%	83.278%	83.490%	83.264%	83.736%	82.382%	83.572%
	PPV	89.194%	91.289%	92.289%	93.973%	94.124%	93.299%	93.943%	93.633%	94.156%	93.636%	94.783%	94.458%
	TNR	94.251%	95.470%	95.927%	96.778%	96.848%	96.501%	97.072%	96.968%	97.207%	96.985%	97.578%	97.352%
	F1S	88.258%	89.313%	90.612%	93.202%	93.032%	93.275%	88.290%	88.271%	88.376%	88.409%	88.148%	88.682%
	Depth Max	2	3	4	4	4	4	2	3	4	5	5	5
	Depth Min	2	3	4	4	4	4	2	2	2	2	2	2

Table 8: Decision tree metrics controlled by maximum depth.

Table 8 shows the same setup as the previous but this time controlled by maximum tree depth. Unlike the max impurity, this hyper parameter seems to really control the tree's capabilities. This is most apparent in the discrete approach. There is definitely a positive correlation between tree depth and classification potential. This is because higher branches have less rows split thus resulting in a higher likelihood for more mixing and thus higher impurity. In other words, low maximum depth gives way to higher leaf impurity and thus lower confidence. As the threshold for the discrete approach reaches above the tree's naturally disposed depth, familiar results begin to appear. Numeric classification does not seem to get too much better with increasing depth, although there is noticeable change when using the entropy minimizer. Although again, it looks like the entropy minimizer is slightly behind on performance compared to the other two in the discrete approach, but interestingly enough it is the reverse for the numerical case just like in the previous table. For the record, an impurity value of 0.1 was used for the threshold in these cases. As a final note, the metrics for the decision tree are all around lower than that of the kNN algorithm.

4.2.1 Principal Component Analysis

Next up, PCA is used to reduce the dataset before training a decision tree on it. The reconstruction corresponds to the k where the recovered percent variance is >= 90%. The results are astonishing. **Table 9** displays the numeric classification of our dataset using PCA vs the numeric classification of our dataset using the original matrix. By the way, discrete analysis is ignored because there are too much variety in values for the tree to be feasible

that way, and maximum impurity analysis is ignored due to lack of effect as shown in the previous tables. Moving on, for comparison's sake, the color limits were kept the same just to show how drastic an improvement the PCA results in. The higher maximum depth seem to reduce the performance of the PCA trees. Misclassification error is definitely the impurity type that yields the best results on all fronts of the PCA. The PCA DT performance is on par if not better than the kNN. Perhaps the reason is because the PCA smooths the data out to in between its original discrete values. However, this result may be due to the PCA being implemented on the dataset before splitting into test and training sets–which means the user should be skeptical since that will likely not happen in reality.

		Max Depth											
		Full Rank						PCA, k = 3					
		2	3	4	5	7	10	2	3	4	5	7	10
Entropy	ACC	92.392%	93.351%	94.491%	94.772%	94.433%	93.842%	97.228%	97.193%	96.848%	96.573%	96.263%	96.146%
	TPR	87.598%	91.966%	92.317%	92.949%	89.725%	87.200%	98.905%	98.948%	97.746%	96.283%	94.813%	94.168%
	PPV	90.470%	88.970%	91.821%	92.081%	94.247%	94.858%	93.489%	93.533%	93.424%	94.007%	94.512%	94.755%
	TNR	94.993%	94.071%	95.643%	95.743%	97.006%	97.440%	96.336%	96.225%	96.375%	96.728%	97.042%	97.207%
	F1S	89.011%	90.443%	92.068%	92.513%	91.931%	90.868%	96.121%	96.164%	95.536%	95.131%	94.662%	94.461%
	Depth Max	2	3	4	5	7	10	2	3	4	5	7	10
	Depth Min	2	3	4	5	6	6	2	3	4	5	7	6
Gini Index	ACC	92.398%	92.906%	93.374%	93.620%	92.825%	93.415%	97.105%	97.193%	96.439%	96.491%	96.327%	96.246%
	TPR	84.895%	85.350%	86.910%	86.724%	84.095%	85.991%	98.565%	98.560%	96.154%	95.919%	95.653%	94.977%
	PPV	92.548%	93.734%	93.545%	94.772%	94.866%	94.721%	93.525%	93.809%	93.816%	94.194%	93.805%	94.314%
	TNR	96.376%	96.948%	96.811%	97.387%	97.541%	97.417%	96.317%	96.446%	96.592%	96.801%	96.682%	96.926%
	F1S	88.556%	89.346%	90.106%	90.570%	89.156%	90.145%	95.979%	96.126%	94.971%	95.049%	94.720%	94.645%
	Depth Max	2	3	4	5	7	10	2	3	4	5	7	10
	Depth Min	2	3	3	3	3	2	2	2	2	2	2	2
Misclassification	ACC	92.205%	92.275%	92.316%	92.398%	92.287%	92.520%	97.070%	97.175%	96.942%	97.135%	97.082%	97.117%
	TPR	83.278%	83.490%	83.264%	83.736%	82.382%	83.572%	98.257%	98.457%	97.869%	98.432%	98.226%	98.088%
	PPV	93.943%	93.633%	94.156%	93.636%	94.783%	94.458%	93.659%	93.676%	93.703%	93.709%	93.725%	93.847%
	TNR	97.072%	96.968%	97.207%	96.985%	97.578%	97.352%	96.434%	96.501%	96.440%	96.435%	96.467%	96.604%
	F1S	88.290%	88.271%	88.376%	88.409%	88.148%	88.682%	95.903%	96.007%	95.741%	96.012%	95.923%	95.921%
	Depth Max	2	3	4	5	5	5	2	2	2	2	2	2
	Depth Min	2	2	2	2	2	2	2	2	2	2	2	2

Table 9: Decision tree metrics comparison of PCA to full rank controlled by max depth.

5 Conclusion

This study involved implementation and analysis of two commonly used supervised classification methods: k-Nearest Neighbors and Decision Trees. They both do the same thing but are very different each with a set of benefits and drawbacks. The kNN is very simple to understand and easy to implement. Also, the results speak for themselves in that kNN consistently performs better than the DT. The issue, however, is that kNN is very computationally heavy, and must be trained every time a new instance is to be classified. For larger datasets this can take more time than its worth, time that a trained and stored Decision Tree classifier does not have to waste. Yes, one of the largest DT pros is that once it has been generated the classifier is simply a series of if statements that can make decisions with a much smaller number of steps than the kNN. The drawback, however is that it is

less accurate (unless PCA is used to offset this) and a bit more complicated to implement because it involves recursion, data types, impurity types, and the list goes on. One more thing on the differences between DT and kNN that would drive one to potentially sacrifice the performance of kNN in favor of DT: DT is interpretable, kNN is not. DT results in a structure with rules that users can familiarize themselves with to increase their expertise in that area; kNN simply throws that knowledge away. Indeed, on top of classification, teaching is perhaps the most useful thing a Decision Tree can do.