Classification Tasks

ML Exercise 2 - Group 19

January 2, 2017

Group	Lukas Stanek	1027203
	Thomas Appler	1025658
	Marten Sigwart	1638152

1 Introduction

Our task was to apply 4 different classification algorithms to 4 different data sets. Hereby, we had to experiment with different parameter settings of the classification algorithm, evaluate the performance by choosing appropriate performance measures and compare results among classifiers and data sets. We also had to perform some pre-processing steps on the data prior to applying the classifiers and study the impact of those pre-processing methods.

1.1 4 data sets

We chose four different data sets for the task. We chose data sets which varied in size, varied in the number of attributes and varied in the number of instances. Further we tried to pick data sets with different numbers of classes. Some data sets had missing values to take into account.

The four data sets we chose are:

- **Letter Recognition** This data set contained attributes about letters shown on a rectangular, black and white display. The classification task is to determine the right letter (A-Z)
- **Internet Advertisements** This data set contained information about images of different web pages. The classification task is to determine whether or not an image is an advertisement.
- **Leukemia** This data set contained medical information of cancer patients. The classification task is to determine whether or not a certain patients has cancer.
- **Congressional Voting** This data set contains information on different voters. The classification task is to determine the party affiliation of the voters.

Each data set will be described in more detail in sections below.

1.2 4 classification algorithms

We chose 4 different classifiers for the task. Hereby, it was important that at least 3 of the classifiers derived from totally different learning algorithms. The classifiers we chose are:

- **K-Nearest Neighbors**
- **Random Forrest**
- **Decision Tree**
- **Bayes Network**

1.3 Performance Measures

Weka calculates quite a few measures to evaluate the performance of a classifier. Probably the most intuitive of these measures is the percentage of rightly (wrongly) classified instances. This generally gives a good first feel of how well the classifier performed. So in our report for comparisons we most often resorted to this measure. Further, in some cases we used F-Measure and ROC Area, which gives some more insight into the general performance of the classifier. F-Measure as a combination of Recall and Precision and ROC Area evaluating the classifier with respect to classifying by chance. Lastly, for comparing how well the classifier did with respect to the different classes, in some specific cases, we looked at the true positive rates for the classes.

2 Data Set 1: Letter Recognition

The objective for this data set is to predict a letter shown on a rectangular, black and white, display. There are 16 numerical features provided for determining the correct letter. These attributes provide information about statistical properties of the letter, like total number of pixels, width, mean of x-axis pixels, etc. All of these values were then scaled into a range from 0 through 15. The 26 capital letters to predict in this set were taken from 20 different fonts. The letters are more or less evenly spread over the 20.000 instances, within a range of 734 to 813 occurrences. This data set contains no missing values. For comparable results 10-fold cross validation was used to measure the algorithms precision. As performance criteria for this data set only the percentage of correctly classified instances will be considered. The reason for this being that if a letter is wrongly classified, it doesn't matter which other letter was recognized instead of the correct one.

2.1 Preprocessing

As the data set was only provided as CSV file without header row, we added a header row for displaying the attribute names in Weka and therefore facilitating the analysis of the results.

2.2 K-Nearest Neighbors

In order to be able to compare the results we always used the same number of neighbors: 1,2,3,4,5,6,8,10,16,20,40,100. The algorithm is generally very fast, even with 100 neighbors, it is usually finished within 30 seconds.

2.2.1 Default implementation

At first we started out using the default KNN implementation of Weka with default settings. The defaults are a *LinearNN* search algorithm with the *Eucledian distance* as distance function. We started out with increasing the amount of neighbors starting at one. Though with this settings we got the best results using just the nearest neighbor.

K	**% Correct**	**% Wrong**
1	95.96	4.04
2	94.925	5.075
3	95.635	4.365

Table 1: Results with default settings

In order to improve the results we started with weighting the neighbors after their distance which helped us further improving the results. The best results where achieved by weighting the neighbors *1/distance*. While the results yielded by the weighting function *1-distance* where also better than without weighting, the were not as good.

K	% Correct	% Wrong
1	95.96	4.04
2	95.99	4.01
3	96.04	3.96
4	**96.115**	**3.885**
5	96.03	3.97
6	96.015	3.985
8	95.75	4.25

Table 2: Results with weighting 1/d

With high values for K the prediction rates get worse.

K	% Correct	% Wrong
40	90.915	9.085
100	84.3	15.7

Table 3: Results for high K's

2.2.2 Manhattan Distance

Using Manhattan distance did not improve our prediction, although again when the neighbors were weighted by their distance the success rate improved.

2.2.3 Chebyshev Distance

Also Chebyshev distance did not improve the result, as it only takes the distance between the attributes which are farthest away from each other as result. As for weighting again, the results improved slightly when using weighted neighbors.

2.2.4 KD Tree Search Method

improves the run time of the classification significantly. The time for evaluating the models with 10-folds cross validation decreased by 66%.

2.3 Random Forest

The default settings for Random Forests are 100 trees with unlimited depth, the number of randomly chosen features per tree is calculated by *log_2(#attributes) + 1*.

2.3.1 Number of Trees

We started out with the default settings and varying the number of trees, starting at 100 trees. The result yielded was already at 96.41% accuracy. Further increasing the number of trees improved the result continuously, but also increased the computation time.

# Trees	% Accuracy	Computation time (s)
50	96.10%	3.1s
100	96.41%	6.55s
200	96.53%	12.33s
300	96.51%	19.13s
400	96.54%	24.46s
500	96.58%	30.66s

Table 4: Results for default settings and different # of trees

Though the performance is improving with the number of trees, the improvements are getting smaller. While an increase of trees from 50 to 100 trees results in a gain of 0.3 percent, the increase from 400 to 500 trees only gains 0.04 percent accuracy.

2.3.2 Increasing Execution Slots

For computing a 10-fold cross-validation at 500 trees on one single thread, the taken time was 5:55min. In order to improve this, the number of execution slots (=threads) can be increased. When repeating the same operation with 10 slots our computation time improved to 1:56min.
Using 50 threads we calculated a model with 1000 trees, which yielded the best result so far with an accuracy of *96.63%*.

2.3.3 Modifying number of features per tree

Through varying the number of features which will be selected per tree, we tried to increase the accuracy. Results for *#Features: 3*:

# Trees	% Accuracy
100	96.51%
200	96.66%
300	96.78%
400	96.75%
500	96.75%

Table 5: Results for 3-feature trees

The best results were achieved with a number of 3 features per tree, other numbers only yielded worse results.

2.3.4 Modifying tree depth

Through modifying the maxial depth a tree is allowed to have we also tried to improve the results of our predictions.We observed that when reducing the depth drastically to something below 10, the prediction rate drops dramatically.

Max depth	Accuracy
3	56.49%
6	73.97%

Table 6: Results below 10

But when raising the depth to 20 - 30 the rates start to normalize again also climb a bit higher than on runs with the default settings.

Max depth	Accuracy
25	96.415%
26	96.42%
27	96.42%
28	96.41%

Table 7: Results above 25

2.3.5 Best combination

The best combination of settings we found was a Tree depth of 28 and the number of features set to 3. The result was a accuracy of **96.79%**.

2.4 Decision Tree - 48J

As for this implementation of the decision tree we also started with the default settings for this data set. The first results were significantly worse compared to the other algorithms. The prediction accuracy was down to 87.92% and also the ROC Area had dropped to 0.954 (The other algorithms were basically at 0.99 or 1). The tree had 1226 leafs and a size of 2451.

2.4.1 Number of objects per leaf

At first we tried to modify the minimum number of objects per leaf. With the default being two we tried to set it to one, which increased the number of leafs to 1771 and the size of the tree to 3541. The result was slightly better but it seemed like over-fitting.
Although increasing the number of minimum nodes per tree resulted in worse results.

Obj. per leaf	**% Accuracy**	**# Leafs**	**Treesize**
1	88.66%	1771	3541
2	87.92%	1226	2451
3	87.20%	1015	2029
4	86.36%	852	1703
5	85.82%	750	1499
8	84.07%	563	1125

Table 8: Results for varying obj per leaf

2.4.2 Pruning

Pruning helps to prevent over-fitting a decision tree, so that it is specialized to one data set and does not generalize well. We adjusted a setting called confidence factor to influence the amount of pruning done during model creation. A higher confidence factor results in less pruning.

Confidence Factor	**% Accuracy**	**# Leafs**	**Treesize**
0.1	87.85%	1138	2275
0.2	87.88%	1198	2395
0.3	87.96%	1255	2509
0.4	87.94%	1255	2509
0.5	87.96%	1267	2533
0.6	87.98%	1303	2605

Table 9: Ajusting the confidence factor

The accuracy is increasing when pruning is decreased, but this would also increase the chances of the tree being over-fitted.
It is also possible to change the pruning mode from C4.5 pruning to reduced-error pruning. It yields slightly worse results than the default pruning method.

2.5 Bayes Network

When using da Bayes Network implementation in Weka we can adjust 2 settings, the *estimator* and the *searchAlgorithm*. With the default settings, which use the K2 search algorithm, the result is an accuracy of *74.31%*. We were able to improve the prediction by increasing the size of parents every network node is allowed to have.

Max # parents	**% Accuracy**
1	74.31%
2	83.69%
3	86.72%
4	86.76%
5	86.76%
10	86.76%

Table 10: Adjusting the maximum number of parents of a node

After using a random order of node in the network the results got slightly better, but unfortunately the random function takes no seed, so that the results vary every run.

Round	**% Accuracy**
1	88.455%
2	88.26%
3	88.46%
4	88.185%
5	87.88%

Table 11: Accuracy of random node order

2.5.1 HillClimber

By using the Hill climber algorithm we could achieve once again better results. But we needed to raise the number of parent nodes agian.

# Parents	**% Accuracy**
1	74.31%
4	89.45%
6	89.45%

Table 12: Accuracy HillClimber search algorithm

We could achieve similar results with the other Hillclimber algorithms *LAGDHillclimber* and *Repeated HillClimber* . They also achieved results of 89% and above.

2.5.2 Tabu search

When using the Tabu Search algorithm we could also improve our results by increasing the number of runs and the length of the tabu list.

# Parents	**# Runs**	**Length Tabu list**	**% Accuracy**
1	10	5	74.31%
4	10	5	84.26%
4	100	10	89.45%

Table 13: Accuracy Tabu search algorithm

So for Bayes Networks and this data set you should be using HillClimber based algorithms as they perform slighlty better than a K2. We could not test the genetic search algorithm as the computation time was to long.

2.6 Conclusion

2.6.1 Letter mix-up

When predicting letters there are some which are pretty well distinguishable from one another and some are not. So it happens that its easy to confuse two letters which look similar to each other. This also applies to our machine learning algorithms. Some letters have a worse prediction rate than others. We can tell which characters are often confused by looking at the confusion matrix. It tells us how often a character was classified correctly or if not for which letter it was mistaken. This allows us also to tell whether the confusion is only going one-way. Meaning that one character is often mistaken for another one, but the other mostly classified correctly.
The top candidates for confusion are:

J	I
P	F
H	K
Q	O

J and I are mistaken for each other equally often, it seems to be generally harder to distinguish these two letters. This also seems to be logical as they look very similar and their geometrical properties could be very similar.
Interestingly Q is often mistaken for O, O is much less often mistaken for Q. The same can be said about P and F, where P is more often mistaken for F, and H and K.

2.6.2 Best Alghorithm

The best algorithm to use for this data set seems to be a random forest with a tree depth of 28, 3 features per tree and a total amount of 300 trees. It results in a total amount of 96.78% of instances classified correctly and a ROC Area of 1. The KNN algorithm is also performing well, with the best value at 96.115% accuracy. Compared to the better ones, 48J and the Bayes Network are not performing that well with peak values at around 87% accuracy.

3 Data Set 2: Internet Advertisements

This data set contains information on different banners/images on websites. The classification task on hand is to determine whether or not an instance is an advertisement or not. A possible use case for this data set would be to remove all adverts from a given website. Some attributes include the geometry of the images (if available), as well as phrases occurring in the URL, the image's URL and alt text, the anchor text and words near the anchor text.

The data set contains a total of 3279 instances, 2821 of which are non-advertisements, 458 are advertisements. Further the data set contains a total of 1558 attributes, 3 of which are continuous variables, the other ones all binary. 28% of instances are missing one or more of the continuous variables.

3.1 Preprocessing

The data set, was contained in two files, one with file ending .name containing all the names of attributes, the other one with ending .data containing the data in a comma-separated format. As Weka prefers data in the ARFF format, we had to do some transformations on the data set prior to loading it into Weka. This was done by first adding the attribute names as header row into the .data file, and saving that file as a CSV file. This file could then be converted by Weka's CSV converter to get the desired ARFF format. Concerning

missing values, we handled them differently for each classifier. We did not normalize the data, as most of the attributes were binary values, and we wanted to keep the proportions of the few numeric values.

3.2 K-Nearest Neighbours

As first classification algorithm we used K-Nearest Neighbors. The parameters we experimented with were the number of neighbors and the distance weighting.

We did not use any missing value replacement strategy for this classifier, as the distance function used by K-NN can handle missing values. When computing the distance between to instances, attributes with missing values are handled in the following way: if the attribute of both instances is a missing values, the distance is handled as zero. If only one attribute is a missing value, maxRange - minRange is returned.

3.2.1 Number of Neighbors:

First, we experimented with the number of neighbors used by the classification algorithm. We did experiments with 1,2,3,4,5,6,8,10,20,50 and 100 neighbors. In the tables below you can see the corresponding results of general performance measures as well as the true positive rates for both classes.

# of Neighbors	% Correctly Classified	F-Measure	ROC Area
1	96.25%	0.966	0.943
2	95.58%	0.956	0.953
3	96.74%	0.966	0.957
4	96.68%	0.966	0.959
5	96.00%	0.958	0.965
6	96.00%	0.96	0.965
8	95.64%	0.953	0.967
10	95.27%	0.949	0.966
20	92.71%	0.916	0.967
50	88.9 %	0.857	0.957
100	88.2 %	0.843	0.959

Table 14: General results for different no. of neighbors

# of Neighbors	TP Rate Ad	TP Rate Non-Ad
1	0.854	0.98
2	0.871	0.97
3	0.804	0.994
4	0.819	0.991
5	0.747	0.995
6	0.763	0.994
8	0.710	0.996
10	0.678	0.998
20	0.486	0.999
50	0.211	0.999
100	0.161	0.999

Table 15: True positive rates for different no. of neighbors

Generally, we can observe that the classification task using K-NN rightly classifies the major part of instances. Depending on the number of neighbors up to almost 97% of instances is classified correctly. The best results seem to emerge when the number of neighbors is fairly low, starting from 1 neighbor we see a slight increase in rightly classified instances up until 3 neighbors. After that the percentage of rightly classified instances starts to decrease again. Looking at the F-Measure our assumption that we find the best classification results by using around 3-4 neighbors is further strengthened as we see F-Measure values of 0.966 for each case. The ROC curve however tells a slightly different story. Here we can we the highest

values for 8-20 neighbors, but the overall the values lie in a range between 0.943 and 0.967 and are therefore quite small.

Another interesting observation is that the true positive rates for class Ad seem to decrease with an increasing number of neighbors where as the true positive rate for class "Non-ad" seems to increase up to 0.999 meaning that almost all "Non-ad" instances are classified correctly. We assume this is because our "ad" instances do not lie all closely together in the data space with respect to their attributes, but rather scattered around the space in smaller groups. Our assumption is that only a few images of class "ad" have very similar attributes, and therefore returning great true-positive rates for a small number of neighbors, whereas with more neighbors, the classifier has to deal with more images that are also quite similar to the given image but of different classes. As we have way more instances of class "non-ad" this comes down to a "majority decision" leaving little to no chance to the classifier but to classify the given instance as "non-ad", therefore explaining the drastic decrease in the true-positive rate for class "ad" and the increase of the same for class "non-ad".

For 2 - 4 neighbors we generally got very good classification results, which is why we think the "ideal" number of neighbors for k-NN lies in that region. However, you cannot choose the ideal number of neighbors without defining a clear idea of what you want to achieve. This may be different depending on the context. For example, looking at the general performance (F-Measure, ROC-Curve) we may conclude that 2-4 neighbors is ideal for the given data set. However, in the context of "ad" and "non-ad" images on web pages, one might want to achieve a high number of identified ads while keeping the number of non-ad images falsely classified as advertisements to a minimum as these might show important information to the user. In this case it might be better to set the number of neighbors to 10 or 20 as this will yield very high true positive rates for class "non-ad" and fairly good true positive rates for class "ad".

3.2.2 Distance Weighting

Next, we experimented with different distance weighting settings. In Weka one can choose between no distance weighting, 1/distance weighting, and 1-distance weighting. Like we discussed in the section before, 2-4 neighbors generally showed "the best" results in the classification, which is why we did the following experiments all using 2-4 neighbors as well as 10 and 100 neighbors just for comparison. Below are the results of the classification task:

Distance Weight	**# of Neighbors**	**% Correctly Classified**	**F-Measure**	**ROC Area**
No distance	2	95.58%	0.956	0.953
1/distance	2	96.43%	0.964	0.964
1-distance	2	96.46%	0.964	0.963
No distance	3	96.74%	0.966	0.957
1/distance	3	96.74%	0.966	0.967
1-distance	3	96.71%	0.966	0.964
No distance	4	96.68%	0.966	0.959
1/distance	4	96.71%	0.966	0.97
1-distance	4	96.71%	0.966	0.966
No distance	10	95.27%	0.949	0.966
1/distance	10	95.73%	0.954	0.973
1-distance	10	95.18%	0.948	0.97
No distance	100	88.2 %	0.843	0.959
1/distance	100	90.18%	0.879	0.972
1-distance	100	88.2 %	0.843	0.963

Table 16: General performance with or without distance weighting

Distance Weighting	# of Neighbors	TP Rate Ad	TP Rate Non-Ad
No distance	2	0.871	0.97
1/distance	2	0.854	0.982
1-distance	2	0.856	0.982
No distance	3	0.804	0.994
1/distance	3	0.808	0.993
1-distance	3	0.800	0.994
No distance	4	0.819	0.991
1/distance	4	0.804	0.994
1-distance	4	0.800	0.994
No distance	10	0.678	0.998
1/distance	10	0.712	0.997
1-distance	10	0.671	0.998
No distance	100	0.161	0.999
1/distance	100	0.305	0.999
1-distance	100	0.161	0.999

Table 17: True positive rates with or without distance weighting

For just 2 neighbors we can see a slight increase in correctly classified instances, also the F-Measure and ROC Area seem to hint at an improved classifier. However, for 3 and 4 neighbors all classifiers seem to show very similar results, so if the distance weighting has an impact, it is now so low that it is basically negligible. We observe the same picture while looking at the results for 10 or 100 neighbors. In general, the F-Measure and ROC Area show very similar results. Only the inverse distance weighting seems to slightly improve the classification on all experiments, which might indicate a more robust method for classification.

So even though the overall performance of the classifier does not seem to be affected that much by the distance weighting we can, however, make out a factor where it does have an impact. In the section above we looked at different numbers of neighbors and we noticed that the true-positive rate for class "ad" seems to decrease when we increase the number of neighbors. We explained this by only a few very similar instances of the class "ad" lying closely together in the data space while there are still quite a few similar instances but of class "non-ad" also in the neighborhood, which leads to some kind of majority count while classifying. The same behavior can also be observed when using distance weighting, though, the decrease of the true-positive rate is not as strong as for no distance weighting. We think this confirms our assumption that the immediate neighbors of an instance of class "ad" are also of class "ad". When using distance weighting within the classifier these instances now get assigned a bigger weight which enhances the probability that the given instance will also be classified as "ad". This reduces the effect of the majority count we observed when not using any distance weighting.

In conclusion, we can say while the distance weighting does not seem to have any strong impact on the general performance, one might still make use of it when the classes are not fairly balanced. E.g. using inverse distance weighting on the classifier might reduce the effect of the majority count when using more neighbors.

3.3 Random Forest

The second classification algorithm we applied was Random Forest. The parameters we experimented with were the number of trees, the number of features, and the bag size. We did not apply any missing value replacement prior to classification as missing values can also be handled by the classifier. This is done by splitting the instance at the branch where the missing value attribute is being compared. So classification is done for all sub instances. The sub instances are then weighted by popularity of the branch.

3.3.1 Number of Trees:

First, we experimented with the number of trees created by the classification algorithm. We did experiments with 2,3,4,5,6,8,10,20,50,100, and 200 trees. We used a bag size of 10 percent.

# of Trees	% Correctly Classified	F-Measure	ROC Area
2	93.69%	0.936	0.885
3	94.75%	0.945	0.914
4	94.88%	0.946	0.93
5	95.09%	0.948	0.944
6	95.09%	0.948	0.956
8	95.12%	0.948	0.962
10	95.46%	0.951	0.968
20	95.67%	0.954	0.977
50	95.7 %	0.954	0.982
100	95.52%	0.952	0.984
200	95.6 %	0.953	0.985

Table 18: General performance for different number of trees

Looking at the results the first thing that catches the eye is that Random Forest classification seems to perform slightly worse than K-NN. For example using 3 trees it classifies 94.75% of instances correctly, whereas K-NN with 3 neighbors classifies 96.74% of instances correctly. Even when increasing the number of trees, the overall classification percentage does not seem to get beyond 96%. As the we increase the number of trees the overall improvement of the classifier seems to decrease, thus stagnating at a certain point. The general performance measures like F-Measure and ROC-Area also show high values, which leads to the assumption that the classifier is robust and does perform quite well. Interestingly, these measures even seem to top the same measures of K-NN with a big enough number of trees, which could mean that Random Forest is a more stable and robust classifier than K-NN and could perform better on different data sets of the same structure.

3.3.2 Number of features

Next, we experimented with the number of features for Random Forest classification. Again we used a bag size of 10, and we used as number of trees 50, as this showed the best result in the experiments before. We tried out 2,5,10,20,50,100,200,500,1000, and 1500 features.

What we observed is that surprisingly the numbers of features used by the classifier does not seem to have such a big impact on the classification results. In general, using more features resulted in more instances being classified correctly, but we could not recognize a specific pattern, nor did we see big differences in the different performance measures. We assume this is because one can distinguish the class of the instance by only looking at a few attributes.

3.3.3 Bag Size:

Increasing the bag size obviously improves the classification results, as more instances are used to build the model. Then the model is tested with data from which the model is built and therefore good classification results occur.

3.4 Decision Tree - 48J

The third algorithm we used was Decision Tree. We experimented with parameters minNumObj, unpruned, confidenceFactor. In the default configuration the tree has at least 2 instances for every leaf (minNumObj), uses confidence factor of 0.25, and is pruned. We did not do any missing value replacement for the same reasons as for Random Forest.

The default configuration returns following results:

% Correctly Classified	TP Rate Ad	TP Rate Non-Ad	F-Measure	ROC Area
96.89%	0.83	0.991	0.968	0.919

Table 19: General performance for default implementation

This is the best result of correctly classified instances we have seen so far on this data set.

3.4.1 Minimum number of instances per leaf

Parameter minNumObj in Weka determines the minimum number of instances per leaf of the decision tree. We tried out values 2, 5, 10 and 20.

Min. # Obj	% Correctly Classified	F-Measure	ROC Area
2	96.89%	0.968	0.919
5	96.58%	0.965	0.924
10	96.65%	0.966	0.923
20	95.73%	0.956	0.885

Table 20: General performance for minNumObj

We can see that increasing the number of minimum instances per leaf, seems to worsen the performance of the classifier. We assume this is because increasing the number introduces further constraints on the final decision tree, which might be forced into forms that are not that suitable for the given data. In this example though it is not of great impact.

3.4.2 Unpruned:

The experiment with an unpruned tree yielded following results:

% Correctly Classified	F-Measure	ROC Area
97.1%	0.970	0.942

Table 21: General performance with an unpruned tree

We can see that not pruning the tree seems to improve the results of the classifier. This is the highest percentage of correctly classified instances we received so far for this data set. However, these results have to be treated with caution as an unpruned tree yields the risk of being over-fitted to the specific data set thus not representing a good general model.

3.4.3 Confidence Factor:

Lastly we experimented with the confidence factor. The confidence factor determines how strongly the decision tree is pruned. Default value is 0.25. The results we got all were quite high with respect to the other classifiers, but none of them were as high as the results we got without using any pruning. In general, more pruning lead to lower classification results, where as less pruning had better classification results while increasing the risk of an over-fitted tree.

3.5 Bayes Network

Lastly, we ran a Bayes Network classifier over the data. Here we experimented with the estimator and the search algorithm.

In Weka one can choose between following four estimators: BayesNetEstimator, SimpleEstimator. In Weka following search algorithms are available: K2, Hill Climber, Genetic Search, LAGDHillClimber, Simulated Annealing, Repeated Hill Climber, Tabu Search, and TAN.

Unfortunately all search algorithms but K2 took too much time to build the classifier model, and were just too much for our machines to handle, which is why we only have results for K2. This is caused by the complexity of the searching algorithm as well as the data set itself, as over 1000 attributes need to be taken into consideration. Below you see the results for both estimators using search algorithm K2.

Missing values are replaced by the mean of the training data.

Estimater	% Correctly Classified	F-Measure	ROC Area
Simple	96.58%	0.964	0.967
BayesNet	96.52%	0.964	0.967

Table 22: General performance with Bayes Network

We can see that both estimator returns very similar results. In comparison to the classifiers we looked at before, K-NN and Random Forest, the Bayes Network Classifier performs strong, but is not extraordinary better than the other two.

3.6 Conclusion

In conclusion we can say that each of the classifiers did a good job classifying the given data set. The reason for this might be that the data's nature is cut out for classifying, in the sense that there exist strong differences between instances of the different classes. All classifiers with reasonable parameters set, returned values of correctly classified instances not lower than 95%. The best classifier out the four would probably be Decision Tree J48 as it was the only one reaching a classification percentage of over 97%. Arguments against it might be the time it took to build the model and the risk of an over-fitted tree. Every time it took around 20 seconds to build the model. In comparison K-NN or Random Forest took much less time to build the model (way under 10 seconds) and still returned fairly good results. Further it must be said that the choice of the optimal classifier not only depends on the data set itself but also on the task the data scientist is trying to accomplish, which might influence the choice of parameter settings to a great deal.

4 Data Set 3: Leukemia

With only 50 instances the leukemia data set has a very low amount of instances but in return has 7130 features which is a really high amount. These 7130 features shall be used to determine whether the given patient has cancer or not.

4.0.1 Preprocessing

As the data set was given in the CSV format and the class as a binary 0 or 1, Weka interpreted the class as numeric and was set to do regression analysis. Via the preprocessing step *NumericToNominal* we converted it to a nominal attribute. Now Weka was doing the right kind of analysis.

4.0.2 K-Nearest Neighbors

We started out modifying the number of neighbors used, which resulted in 1 neighbor as being the best setting. When we added distance weighting to the equation we got the same accuracy for four neighbors as for one. We also increased our ROC Area. But we lost some at false positives and the f-measure.

# Neighbors	Weighting	Accuracy	False Pos.	F-Measure	ROC Area
1	-	84%	0.236	0.835	0.802
2	-	78%	0.391	0.748	0.805
4	-	82%	0.320	0.801	0.940
4	1/d	84%	0.284	0.826	0.967

Table 23: KNN results

Afterwards we used the Manhattan distance function for the algorithm and results were improving. We could improve our accuracy to 88% with the settings set to a single neighbor. The other settings just resulted in the same 84% accuracy as before and additionally classified a big amount of cancer patients as healthy ones. Specifically in one run 8/18 patients with cancer were predicted to have none. When thinking about giving wrong diagnosis to patients it would be better to wrongly diagnose him with cancer, so that it can be double checked. But when you diagnose him to be healthy instead he won't do any follow up checks. So our single neighbor Manhattan distance algorithm was the only one which could reduce the amount of these wrong predictions to 5/18 and also had the best accuracy.

4.0.3 Random Forest

As usual we started out with the default settings for Random Forests in Weka. This meant 100 trees with the number of features per tree set to the integer value of *log2(#attributes)+1* , which equals 13. We started out modifying the number of trees and features.

# Trees	# Features	Accuracy	F-Measure	ROC Area
10	default	80%	0.783	0.892
50	default	86%	0.850	0.951
100	default	86%	0.850	0.957
200	default	86%	0.850	0.964
50	200	94%	0.969	0.986
100	**200**	**98%**	**0.980**	**0.993**
200	200	98%	0.980	0.988
300	200	96%	0.960	0.986
100	300	94%	0.940	0.991
100	400	94%	0.939	0.990
50	1000	98%	0.980	0.988

Table 24: Results with different number of trees and features

While experimenting with number of features and trees we found that the best combination would be using 200 features an 100 trees as this would result in an 98% accuracy with only one misclassification. Also it results in the best ROC and F measure values. We also experimented with the tree depth, but this had no impact on our results with this data set.

4.0.4 Decision Tree - 48J

The most interesting observation here is that although the number of attributes is very high, the classifier is building a very small decision tree. In most cases a tree with just two or three leaves is built. This means that there is at least one attribute, which has a high correlation with the class and separates the data well. And in fact, with the default settings, the classifier is building a tree with three leaves and a prediction rate of 82%, containing the attributes *U46499_at* and *U06233_at*. After setting the minimum number of objects per leave to 7, a two-leaved tree with an even better prediction rate was built (86%), which uses just the *U46499_at* attribute. And after setting the minimum number of objects per leaf to 20, we got an even better precision rate of 92%. This time, the tree also just contained one attribute: *M23197_at*. Interesting here is also, that the TP of the class leukemia quals 1 rate is 1. This means, that all cases with leukemia were predicted right, but also with some wrong false positives. This means, that this two attributes *M23197_at* and *U46499_at* are highly correlated with the class attribute and separate the data well. Changing other parameters than the number of objects per leaf do not have an impact on the result. The tree is always build the same way.

% Accuracy	F-Measure	ROC area	attributes used	leafs of tree
82	0.821	0.811	U46499_at, U06233_at	3
86	0.862	0.852	U46499_at	2
92	0.921	0.913	M23197_at	2

Table 25: Results for default settings and different # of trees

4.0.5 Bayes Network

We started out with the default settings of a Bayes Net and it then changed the search algorithms. We changed the number of parents each node is allowed to have in order to improve the results, although the results were already at 98%. Creating the nets with a higher amount of parents are taking a long time to compute.

Search algorithm	# Parents by nodes	Accuracy
K2	1	98%
K2	2	98%
Hill Climber	1	98%

Table 26: BayesNet results

The results did not improve on any modification we did on the settings. The were always returning a 98% accuracy, F-Measure at 0.980 and a ROC Area of 0.972.

4.0.6 Conclusion

It seems, that the best algorithms to use for this data set would be a random forest with 100 trees and 200 features per tree. Also a bayes network could be used as it has similar prediction qualities, although in our experiments the bayes networks would need much more time to compute and would often let Weka run out of memory.

5 Data Set 4: Congressional Voting

This data set contains 16 votes on different topics (like immigration, education spending) for each of the US House of representatives Congressmen. Those 16 votes were identified by the CQA as the 16 key votes. The CQA lists nine different types of votes, which are simplified here to *yea* (voted for, paired for, announced for), *nay* (voted against, paired against, announced against) and *unknown* (voted present, voted present to avoid conflict of interest, no vote or unknown position). The objective here is now to predict, whether a person is republican or democrat. So in total, the data set is quite small and contains 18 attributes (*id*, *class* and 16 topics) and 218 instances.

5.1 Preprocessing

First we removed the attribute *id*, due to the fact that attributes with unique values do not contain useful information. The second step was to replace all values *unknown* with *?*, so that Weka is able to interpret this as missing values, and not as nominal values. To treat the missing values, we used the preprocessing command *ReplaceMissingValues*, which replaces every *?* with the mode, that means with the value which appears the most. Another approach can be to first transform all nominal values to numerical zeros and ones and perform then the ReplaceMissingValues command. This way, all missing values are not replaced with the mode of the attribute, but with the mean. This means that these values may be between zero and one. But it turned out, that this approach decreases our prediction accuracy in most cases, therefore we will not use this method. Another approach could be also to remove all instances with missing values. But in this case it does not make much sense, because the rate of missing values is quite high and the size of the set is very small. Therefore we would eliminate too much instances.

5.2 kNN

5.2.1 parameter k

With default setting we first tested the data set with an 10-fold cross validation with different values for k and it turned out, that three gives us the best accuracy rate of 93.12%. But also with values 6 and 7 for k the results are quite good. In general it looks like every value for k works quite good for this data set. Although the prediction rate gets worse with high values for k, the decrease is not that big. For example if k is 40, the accuracy decreases just about 3 percent. Also for k equal 1 the result is very good (91.74%). Considering that that data set is very small, this means that just 3 instances of 218 were misclassified more than with k equals 3. It looks like, that except some outliers, the data is relatively good separated, because about 92% of the instances have a nearest neighbor with the same class. An interesting observation is also that the weighted ROC area stays very high, also for high values for k. For example for values 16-40, the weighted ROC area is even higher, that the ROC area with k equals 3. After that we also tried the two

different weighting approaches, but they had no impact on the results. All results with different k's are shown the following table.

k	% Accuracy	F-Measure	ROC area
1	91.7431	0.918	0.955
2	91.7431	0.918	0.962
3	93.1193	0.932	0.974
4	91.7431	0.918	0.979
5	91.2844	0.914	0.981
6	92.2018	0.923	0.972
7	92.2018	0.923	0.973
8	91.7431	0.918	0.976
10	91.2844	0.914	0.976
16	91.2844	0.914	0.981
20	91.2844	0.914	0.982
40	89.9083	0.900	0.981
100	88.0734	0.882	0.969

Table 27: Results for different values for k

5.2.2 Distance Functions and Search Methods

Except the *Chebyshev* distance function, all distance functions lead to the same result. With the *Chebyshev* distance the accuracy got a lot worse and decreased to 72.48%. When weighting the the neighbors, the *Manhattan* distance function decreased the precision rate, but just to 90.82%. Due to the fact that there are just 218 instances, other search methods that thee default one do not really have a noticeable effect on performance.

5.3 Random Forest

5.3.1 Number of trees

With default settings we first started to vary the number of trees and it turned out, that 16 gave us the best result of 97.25% accuracy and the lower the number is, the worse the result. As expected, because of the small data set the ideal number is quit low. But also with a higher number of trees the classifier yielded almost the same result (96.789% accuracy). But this difference is not significantly worse, because just one instance more was misclassified. This means that increasing the number of trees improves the result, until a point, where it stays the same.

num. trees	% Accuracy	F-Measure	ROC area
100	96.789	0.968	0.996
50	96.789	0.968	0.996
20	96.789	0.968	0.995
17	96.789	0.968	0.994
16	97.2477	0.973	0.994
14	95.8716	0.959	0.994
8	95.4128	0.968	0.992

Table 28: Results for default settings and different # of trees

5.3.2 Number of Attributes

The default number of attributes until now was 5 (*log2(#attributes)+1*). After testing with different numbers, it turned out, that just very low values (<4) and high values (>14) lead to worse results. Others in between lead to almost the same result.

num of. att.	% Accuracy	F-Measure	ROC area
16	95.8716	0.959	0.987
15	95.8716	0.959	0.987
14	96.789	0.968	0.989
12	97.2477	0.973	0.992
10	97.7064	0.977	0.989
8	97.2477	0.973	0.992
5	97.2477	0.973	0.992
4	96.789	0.968	0.995
3	96.3303	0.963	0.989
2	94.9541	0.950	0.992

Table 29: Results fpr different numbers of attributes

5.3.3 Tree depth

In the default settings, an unlimited depth of trees is possible. As expected, limiting the maximal depth of trees did not improve our results. If the maximal tree depth is smaller than 8, the result is worse, if it is 8 or more, the result stays the same.

max depth	% Accuracy	F-Measure	ROC area
2	96.3303	0.963	0.990
4	96.789	0.968	0.991
6	97.2477	0.973	0.988
8	97.7064	0.977	0.989
20	97.7064	0.977	0.989
100	97.7064	0.977	0.989

Table 30: Results for different tree depths

5.4 Decision tree

5.4.1 Pruning

First we compared the results of the decision tree with and without pruning. We used the default settings and gained an increase of 1.38% accuracy with pruning. Next we tested the parameter *confidence factor*, which is used for pruning. It turned out, that with a confidence factor from 0.25 to 0.5, the result stays the same. But if this factor is less than 0.25 or higher than 0.5 the result gets worse, although the difference is not that big. So in our case, pruning improves the results, but not significantly. Usually the problem of an unpruned tree is, that it is too specialized. But in our case, this is not really the case. Just 3 instances more are misclassified, than with an pruned tree.

pruned	confidence fac.	% Accuracy	F-Measure	ROC area
no	-	95.4128	0.954	0.969
yes	0.1	96.3303	0.963	0.943
yes	0.25	96.789	0.968	0.952
yes	0.50	96.789	0.968	0.952
yes	0.55	95.4128	0.954	0.969
yes	0.70	95.4128	0.954	0.969

Table 31: Results for pruning with different confidence factors

5.4.2 Other parameters

Changing other parameters like *collapseTree*, *useLaplace* or *numFolds* had no impact on the results in this case. The only other parameter which made a noticeable difference was *minNumObj*. If this parameter is

not set to 2 or 3, the results get a bit worse. Interesting here is also, that if we set the parameter to 10, a very small tree with just one attribute(*physician-fee-freeze*) is built. But still, the prediction rate decreases just a bit (96.33%). This means that this attribute *physician-fee-freeze* is a high correlated with the class attribute.

5.5 Bayesian Network

5.5.1 Search algorithm

First we started to test the classifier with different search algorithms. With the default search algorithm *k2*, we gained an accuracy of 89.9% and it turned out, that other methods like *hillclimbing*, *repeated hillclimbing* and *tabu search* lead to exactly the same result. The best result is yield with the *TAN*-search, which increased the accuracy rate to 95.41%. Also the method simulated annealing leads to good results, but the performance is a lot worse. The runtime increased from about 0.02 seconds to almost two seconds.

Search algo.	**% Accuracy**	**F-Measure**	**ROC area**
K2	89.9083	0.900	0.977
LAGDHillClimber	90.8257	0.909	0.982
SimulatedAnnealing	94.4954	0,945	0.986
TAN	95.4128	0.954	0.992

Table 32: Results with different search algorithms

5.5.2 Other parameters

We also tried out other estimator functions, but it turned out that except the *MultiNominalBMAEstimator* function all lead to the same result. *MultiNominalBMAEstimator* just decreases the prediction rate dramatically, which is obvious because the set contains binary nominal data, and not *multinominal* data. After that we also tested the *useADTree* parameter, but it also had no impact on the result.

5.6 Conclusion

In general, all classifiers worked relatively well on this data set and had a prediction rate over 90%, most of them even over 95%. It seems, that except of some outliers, the data of the set is well distributed. We have seen, that for example the attribute *physician-fee-freeze* is high correlated with the class attribute. Another sign for this is the outcome of the *kNN* tests. There we have seen that almost 92% of all instances have a closest neighbor with the same class. And also with a greater value of k, the results do not change dramatically. Also the difference between a unpruned tree and a pruned tree is not that big, which means that even the unpruned tree gives us a good generalisation of the data.

6 Conclusion

6.1 KNN

Although the NKK classifier is very simple, its results are quite good and in most cases almost as good as other classifiers tested or even better. Most sensitive parameter here is k, which is always a small (<10) number in our cases and increasing it leads to worse results. Also setting k to one is suboptimal. It just yields a good result, if the data is good distributed and every instance has a closest neighbor with the same class. We also saw, that weighting the the neighbors most of the time improves the result. Only with data set 4 it was not the case. About the different distance functions we can say, that the *Euclidian* and the *Manhattan* distance functions gave us the best results. It depends on the data which function is better, but in general the difference is not that big. We also saw, that the *Chebyshev* distance function always gives us a lot worse result, since it takes the farthest neighbour and calculates the distance.

6.2 Decision tree

Probably the most important step that the classifier does, is to prune the tree. In all cases (except data set 3) pruning the tree improves the results. Without pruning, the tree is often too specialized and does not generalize the data well. For pruning, the *j48* implementation uses the parameter *confidence factor*, which optimal value is dependent on the data. We saw that for letter recognition, increasing the value improves the results, but we also saw, that it can decrease the result for other data (data set 4).

6.3 Random forest

Probably the most important parameter here is the number of trees generated. Generally we can say, that increasing the number of trees is also improving the prediction rate. But at some point, either the improvement of the results stop (data set 4) or the improvements get very small (data set 1). Because of performance issues, therefore for a very big data set, it is sometimes not worth to increase the number of trees for a small improvement. Another important parameter here is the number of features used per tree. We saw, that the default value (*log2(#attributes)+1*) for this generally gives quite good results. We also recognized, that very small values like 1 or high values like the number of attributes of the set is not optimal and lead to worse results.

6.4 Bayes Network

The parameter with the most impact on the classification result is the search algorithm used. Between all search algorithms we saw that most of them performed similarly well, even though we had some that performed generally worse than others (K2) and some that for some data sets performed a lot better than others (Simulated Annealing and TAN in data set 4). Although returning good classification results on most data sets, the computation time of the classifier is a big downside, as the models of all search algorithms except K2 take a long time to build, in some cases (data set 2) just too long to complete the computation.