Business Intelligence 2017 - Assignment 3

Data Mining - Group 20

Michael Bauer
e01351159@student.tuwien.ac.at

Solomon Kahsai
e01326114@student.tuwien.ac.at

Johannes Sederl
e01225759@student.tuwien.ac.at

1 DATASET

We selected the datasat "har"[1] (Human Activity Recognition). It is a sample from a HAR database built from the recordings of 30 subjects performing acitivies of daily living while carrying a waist-mounted smartphone with embedded inertial sensors.

The sensor signals were pre-processed and from each window, a vector of features was obtained by calculating from the time and frequency domain. This resulted in a list of 561 features. The experiments were recorded on video-tape to label the data after recording manually.

The dataset has the following characteristics:

- Instances: 10299
- Attributes: 562
- Class labels: 6

The dataset contains no missing values and all features are numeric with a value range of -1 to 1. No outliers could be identified. The class labels and corresponding movements are the following:

- 1: WALKING
- 2: WALKING_UPSTAIRS
- 3: WALKING_DOWNSTAIRS
- 4: SITTING
- 5: STANDING
- 6: LAYING

The distribution of the classes among the instances of the dataset is visualized in Figure 3, where the classes 1 to 6 correspond to the differently colored bars from left to right.

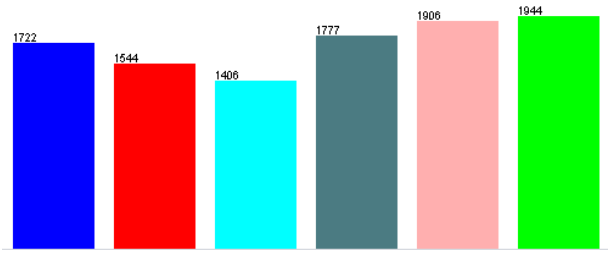

Figure 1: Class distribution among instances of the dataset

We want to emphasize that prior to analysis and experimenting with different algorithms we expected that classification of instances to sitting and to standing would be difficult and could easily be swapped as the position and sensor readings was quite similar. In the course of experimenting with different algorithms our expectations were proven to be true.

[1] https://www.openml.org/d/1478

2 CLASSIFICATION

2.1 Algorithm selection

By running several classification algorithms (K-Nearest Neighbor, NaiveBayes, RandomForest, Support Vector Machine) and by analyzing the dataset we identified the best two scoring ones (KNN and Support Vector Machine) in terms of correctly classified instances. These two are described in further detail below.

2.2 Preprocessing

No preprocessing was required for running the classification algorithms. All attributes had the same value range and were in the same numeric format having up to six decimal places.

2.3 Subsampling

We used an Intel core i7 processor on a Window's 8GB RAM notebook. Although the dataset is of high dimensionality (561 features) and a high number of instances (10299) the required algorithms finished within a maximum time frame of five to seven minutes. This is why we did not consider subsampling being necessary. However, for part C) of this assignment we recommend reducing the size of the dataset for performance reasons. We had difficulty dealing with the large size when replacing missing values.

2.4 Training & Testing

2.4.1 K-Nearest Neighbor. The k-Nearest Neighbor algorithm (=knn) classifies objects based on a majority vote of its k-nearest neighbors. By choosing a k=1 each object is assigned to the class of its closest neighbor. By increasing k the effect of noise is diminished. However, by choosing a high k the boundary between two or several classes becomes less distinct.

Parameters. We started with the standard WEKA parameter of the IBk-algorithm (=knn-algorithm) k = 1 and with EuclideanDistance as the distance function. The standard setting of EuclideanDistance is to normalize all value ranges. For testing reasons we also selected k=2, but if the algorithm checks the two closest neighbors and evaluates the majority class, it is possible that no majority class can be found. We continued with testing odd numbers for k and found best scoring parameters for k=3 and k=11. By testing k=1000 and k=5000 we could see that the boundary between classes was not very clear anymore.

```
weka.classifiers.lazy.IBk -K 1 -W 0 -A
"weka.core.neighboursearch.LinearNNSearch -A
"weka.core.EuclideanDistance -R first-last"
```

Figure 2: Command for knn with k = 1

a	b	c	d	e	f	<– classified as
1719	1	2	0	0	0	| a = 1
0	1541	3	0	0	0	| b = 2
4	5	1397	0	0	0	| c = 3
0	3	0	1651	125	0	| d = 4
0	0	0	143	1763	0	| e = 5
0	0	0	1	0	1943	| f = 6

Table 1: Confusion Matrix for k = 1

k	Corr. cf. instances (%)	Precision (mean, %)	Recall (mean, %)	Mean Absolute Error
1	97.2327	97,2	97,2	0.0094
2	95.8054	96.2	95.8	0.0133
3	97.1259	97.1	97.1	0.0161
5	96.9317	96.9	96.9	0.02
7	96.7764	96.8	96.8	0.0228
11	97.1356	97.1	97.1	0.016
13	96.7764	96.8	96.8	0.0228
100	92.3488	92.4	92.3	0.061
500	88.1445	88.5	88.1	0.0994
1000	85.785	86.5	85.8	0.1254
5000	54.112	?	54.1	0.2305

Table 2: Effect of different k-settings

We tested Manhattan Distance as a different distance function. Compared to Euclidean Distance at k=3 it classified a higher share of instances correctly. However, we continued with the standard setting of WEKA's IBk algorithm: Euclidean Distance.

Correctly classified instances	10125	98.3105 %
Incorrectly classified instances	174	1.6895 %
Mean absolute error	0.0108	
Root mean squared error	0.718	
Relative absolute error	3.8831 %	
Root relative squared error	19.2863%	
Total number of instances	10299	

Table 3: Summary of k = 3, Manhattan Distance

Scaling. Scaling the attributes to a range of 0 to 1 (before: -1 to 1) with scaling filter 'weka.filters.unsupervised.attribute.Normalize-S1.0-T0.0' does not have any effect on the result of knn with k=3 as the knn-algorithm normalizes the data by default to a range of 0 - 1.

The same is the case for standardization, run by the following filter: weka.filters.unsupervised.attribute.Standardize

After deactivating the standard-parameter of normalization by Euclidian distance and running the knn-algorithm with k=3 we could observe a faster execution and a slight decrease in the number of correctly classified instances:

Moreover, we tested a - in our eyes - non-useful scaling to a range of -15 to 35. We observed a minor increase in the percentage of correctly classified instances (from 97.1259% to 97.1356%). We suspect that especially instances classified as '4' (sitting) or '5' (standing) lie very close to each other. By enlarging the scale, the distance between relevant attributes increases and classification improves by some thousandth parts.

Correctly classified instances	9965	96.757 %
Incorrectly classified instances	334	3.243 %
Mean absolute error	0.0193	
Root mean squared error	0.0971	
Relative absolute error	6.9621 %	
Root relative squared error	26.078%	
Total number of instances	10299	

Table 4: Summary of k = 3, mean/unit variance, not normalized

Correctly classified instances	10004	97.1356 %
Incorrectly classified instances	295	2.8644 %
Mean absolute error	0.016	
Root mean squared error	0.0903	
Relative absolute error	5.7866 %	
Root relative squared error	24.2521%	
Total number of instances	10299	

Table 5: Summary of k = 3, scale: -15 to 35, not normalized

Training Test splits. As we could not observe a major increase in accuracy by different scalings we continued with k=3 for evaluating the effect of different training and test set splits on accuracy and performance.

% Train split	Corr. class. (%)	Mean abs. error	Test time (s)
1	78.8348	0.0893	4.26
5	89.0127	0.0523	13.03
10	91.6172	0.0393	21.76
20	93.4944	0.0309	37.64
30	94.923	0.0254	50.47
40	95.7113	0.0227	58.31
50	96.1546	0.0208	55.63
60	96.7476	0.0189	51.94
70	97.0874	0.0175	44.93
80	97.6214	0.0158	29.27
90	97.767	0.0153	16.54
99	100	0.0108	1.72

Table 6: Effect of different train splits

The increase in correctly classified instances increases on a diminishing scale. While it only takes almost 10% of training data to already achieve a classification accuracy of above 90%, a further increase to about a third of the total instances for training is needed for reaching 95% of correctly classified instances. Also testing takes in total four times longer than at a level of using about 1000 instances for training.

Best parameters. For performance reasons we selected k=3 and Euclidean Distance for the following experiments. We used the following WEKA command: weka.classifiers.lazy.IBk -K 3 -W 0 -A "weka.core.neighboursearch.LinearNNSearch -A ẅeka.core.EuclideanDistance -R first-last¨
It showed slightly worse classification results than k=11, but gave faster results. In the visualization Figure 3 one can observe easily the two classes (4 and 5) where most misclassifications occur. The x-axis shows classifications as contained in the dataset, wheras the labels on the y-axis denote the classifcation by knn-algorithm.

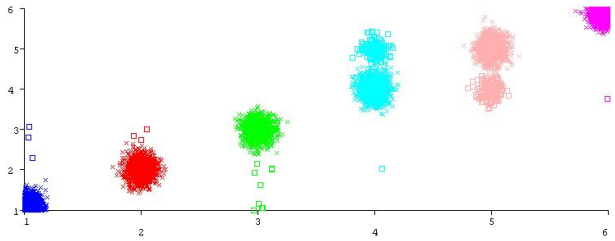

Figure 3: Class distribution among instances, knn=3

Correctly classified instances	10003	97.1259 %
Incorrectly classified instances	296	2.8741 %
Mean absolute error	0.0161	
Root mean squared error	0.0904	
Relative absolute error	5.806 %	
Root relative squared error	24.2735%	
Total number of instances	10299	

Table 7: Summary of k = 3, Euclidean distance

2.4.2 Support Vector Machine. We chose Support Vector Machine as the second algorithm for our dataset, where we used the WEKA implementation SMO. With SVM, support vectors are created to seperate the data points of a dataset to allow classification. Besides linear classification, SVM is also able to classify data non-linearly by projecting the data space into a higher dimensional space with a so-called kernel.

Parameters. In SMO, different parameters can be set to alter the behaviour of the algorithm. We adapted the kernel used by SMO, the tolerance parameter and the complexity parameter to get the best possible results for our dataset.

At first, we examined the impact of using different kernels for the classification of the dataset. We therefore tested `PolyKernel`, `NormalizedPolyKernel`, `RBFKernel` and `Puk`, which are provided by WEKA. Interestingly, the `RBFKernel` and `Puk` took extraordinarily long to build their models (> 20 mins), so because of performance reasons we decided to not take that specific kernels into further consideration for our dataset.

In the respective tables 9 and 8, the results for the `PolyKernel` and the `NormalizedPolyKernel` with their standard configurations are shown.

Correctly classified instances	9956	96.6696 %
Incorrectly classified instances	343	3.3304 %
Mean absolute error	0.223	
Root mean squared error	0.3115	
Relative absolute error	80.4705 %	
Root relative squared error	83.6807%	
Total number of instances	10299	

Table 8: Summary of NormalizedPolyKernel

Correctly classified instances	10147	98.5241 %
Incorrectly classified instances	152	1.4759 %
Mean absolute error	0.2226	
Root mean squared error	0.3107	
Relative absolute error	80.3164 %	
Root relative squared error	83.4668%	
Total number of instances	10299	

Table 9: Summary of PolyKernel

As it can be seen in the result tables (table 9 and 8), the `PolyKernel` achieved significant better results than the `NormalizedPolyKernel`, so we used the `PolyKernel` for all further calculations.

After choosing the optimal kernel for our dataset, we examined the impact of the tolerance parameter. The default value for the tolerance parameter in SMO is 0.001, so we tried variations with lower and higher values. However, after some test runs, we observed that this parameter has not a big impact on the classification results. The differences were only minor, so we concluded that this parameter is irrelevant for further optimizations and we used the default value (0.001) for all further examples.

Next parameter to be considered for optimization is the complexity parameter, which has a default value of 1.0 in SMO. We tried different variations of this parameter (0.1, 10, 15, 20 and 100) and concluded that the optimal value for the complexity is 15, which offered the best results for our dataset. Table 10 shows the results of with the complexity parameter set to 15.

Correctly classified instances	10174	98.7863 %
Incorrectly classified instances	125	1.2137 %
Mean absolute error	0.2225	
Root mean squared error	0.3106	
Relative absolute error	80.2953 %	
Root relative squared error	83.4407%	
Total number of instances	10299	

Table 10: Summary for C=15.0

In conclusion, for optimal results with SMO a `PolyKernel` was used, the tolerance parameter set to the default value and the complexity parameter set to 15. Table 11 shows the results of the classification and table 12 shows the confusion matrix for the optimal configuration of SMO for our dataset.

Scaling. Due to the fact that our dataset already consists of values in the interval [-1;1], the normalization to [0;1] has no impact on the

Correctly classified instances	10174	98.7863 %
Incorrectly classified instances	125	1.2137 %
Mean absolute error	0.2225	
Root mean squared error	0.3106	
Relative absolute error	80.2953 %	
Root relative squared error	83.4407%	
Total number of instances	10299	

Table 11: Summary results for optimal SMO configuration

a	b	c	d	e	f	<– classified as
1720	1	1	0	0	0	| a = 1
1	1543	0	0	0	0	| b = 2
1	0	1405	0	0	0	| c = 3
0	1	0	1723	53	0	| d = 4
0	0	0	67	1839	0	| e = 5
0	0	0	0	0	1944	| f = 6

Table 12: Confusion matrix for optimal SMO configuration

overall results. Also the standardization of the training data, where a mean of 0 and a variance of 1 is achieved, shows no improvement on the results. Table 13 shows the result with standardization of the training data.

Correctly classified instances	10129	98.3494 %
Incorrectly classified instances	170	1.6506 %
Mean absolute error	0.2226	
Root mean squared error	0.3109	
Relative absolute error	80.3312 %	
Root relative squared error	83.5158%	
Total number of instances	10299	

Table 13: Summary results for standardization of training data

We therefore concluded that no normalization or standardization is needed to achieve optimizations for our dataset.

% Train split	Corr. class. (%)	Mean abs. error	Test time (s)
1	87.8089	0.2252	21.55
5	94.767	0.2235	15.36
10	95.9435	0.2232	16.49
50	98.2909	0.2226	18.85
99	99.0291	0.2224	16.72

Table 14: Effect of different train splits

Training Test splits.

3 MISSING VALUES

We used Java to create the missing values according to the requirements of the assignment. The original data has no missing values. We used two algorithms. One to create missing values over the entire data set and another over the highest and lowest information gain attributes. We have already found out what the highest and lowest information gain attributes were from our runs on the original file. The settings we used on WEKA were as follows: attribute evaluator (InfoGainAttributeEval) and ranker (threshold: -1.7976931348623157E308 and number to select at -1). The highest information gain attribute was V303, while the lowest was V189. 10 percent is assumed to be small and 90 percent large when it comes to how we chose our missing value proportions. A snapshot of the WEKA output is displayed in Figure 4.

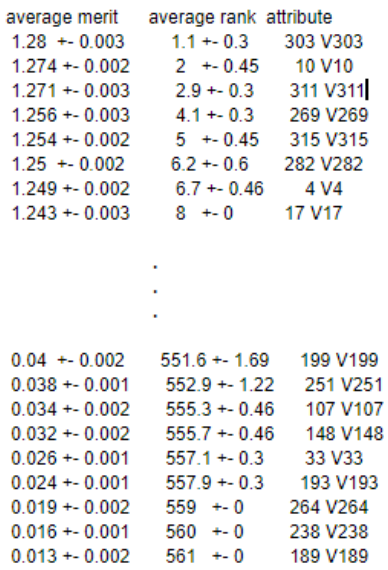

Figure 4: Snapshot of WEKA output on average merit of attributes

3.1 Script

The script used to generate the missing values for all the datasets

The original arff file is opened and a loop starts over the the size of the data set. Then the first line is held and over a second loop which loops over the size of the single line, a condition of whether or not a random number that was generated is less or equal to a threshold which we have named OVERALL_PERCENTAGE_MISSING is checked, if so, a missing value denotation is inserted at the location it was found in the line. We used a seed of 2 for the random number generator. This was arbitrarily chosen. The threshold is at 0.9 for generating what we call large missing values and at 0.1 for small missing values

The script used to generate the missing values for low and hight information gain attributes

The algorithm is similar to the above one except for the fact that

it uses a MAP called MISSING_PERCENTAGES_FOR_COLUMNS to hold the column in question (high or low information gain attribute) and the percentage of missing value (large 0.9 or small 0.1 missing values). After it opens the file, it goes into a loop over the size of the data set, it holds on to the first line and for each of the MISSING_PERCENTAGES_FOR_COLUMNS map checks if the random number that is generated is less or equal to the missing value threshold (in our case 0.1 for small and 0.9 for high) that was given to the map, if it is, change the value of the line instance at that location with the missing value denotation. Again, as with the missing value for all the data set generator we explained above, We used a seed of 2 for the random number generator. This was arbitrarily chosen. The threshold is at 0.9 for generating what we call large missing values and at 0.1 for small missing values.

Based on the described scripts we generated six modified datasets (MV = missing values):

- output_largeMValuesOnHighInfoGain.arff (attribute v303)
- output_largeMValuesOnLowInfoGain.arff (attribute v189)
- output_smallMValuesOnHighInfoGain.arff (attribute v303)
- output_smallMValuesOnLowInfoGain.arff (attribute v189)
- output_largeMVOnAllDataSet.arff
- output_smallMVOnAllDataSet.arff

3.2 Strategies for dealing with missing values

We implemented three different strategies for dealing with missing data:

- a) Ignoring the respective attributes
  In order to ignore the selected attributes (v303 and v189) we removed them for each experiment in the preprocessing step in WEKA.
- b) Replacing the missing attribute values by the mean value of that attribute in the entire dataset
  For this, we used WEKA's reprocess filter 'ReplaceMissingValues' which replaces all missing values for numeric attributes in a dataset with the mean of the attribute in the entire dataset.
- c) Replacing the missing attribute by the mean value of that attribute in the respective class
  To our knowledge WEKA does not provide any standard filter for the replacement of missing values with the mean per attribute per class. We used R for data manipulation (see replacingMissingValuesWithMean.r), which reads the ARFF-file, calculates the mean of a specified attribute per class, goes through a data frame containing the dataset and replaces column-specific missing values by the corresponding class mean. Then it writes the ARFF-file back to the same location. In the R-file the location of the .ARFF-file containing missing values and the attribute with missing values need to be specified.
  We had difficulty in adapting our column-specific algorithm to the datasets containing missing values distributed accross all attributes and were not able to experiment with these.

Table 15 gives an overview of experiments on different datasets, followed by our analysis and findings:

When analyzing attributes V303 and V189 separately, we could observe that substituting missing values with the attribute mean of the class (c) yields at least slightly higher results than strategy (b). To us this is comprehensible, as taking the mean across all values of the attribute is in this case less accurate than replacing missing values with the mean of the corresponding class.

However, one finding was interesting: Ignoring an attribute with a higher information gain (V303) produced a better result than ignoring the last-ranked attribute. We suspect the reason for this lies in the confusion of classes 4 and 5 and a stronger influence of V303 on this confusion than of V189. This becomes obvious in comparing Table **??** and **??**.

Attribute	% Missing	Strategy	Corr. classified %
V303		a)	97.1259
V303	10	b)	97.1065
V303	10	c)	97.1356
V303	90	b)	97.1259
V303	90	c)	97.1551
V189		a)	97.0677
V189	10	b)	97.1162
V189	10	c)	97.1162
V189	90	b)	97.0774
V189	90	c)	97.0871
all	10	b)	95.5724
all	90	c)	35.217

Table 15: Results from missing-value-experiments

a	b	c	d	e	f	<– classified as
1718	2	2	0	0	0	| a = 1
1	1541	2	0	0	0	| b = 2
5	5	1396	0	0	0	| c = 3
0	3	0	1630	141	3	| d = 4
0	0	0	131	1775	0	| e = 5
0	0	0	1	0	1943	| f = 6

Table 16: Confusion Matrix for ignoring V303

a	b	c	d	e	f	<– classified as
1717	3	2	0	0	0	| a = 1
1	1541	2	0	0	0	| b = 2
5	5	1395	0	0	0	| c = 3
0	3	0	1629	142	3	| d = 4
0	0	0	134	1772	0	| e = 5
0	0	0	1	0	1943	| f = 6

Table 17: Confusion Matrix for ignoring V189

For both attributes strategy b) had an equal or even positive effect on classifier performance when compared to simply ignoring the attributes as a whole. The reason for this is that both attributes

have an information gain greater than 0.
Interestingly for attribute V303 we could observe that by applying strategy c) the highest number of instances could be correctly classified, even greater than in a setting without any missing values (see section 2).
The classifier performance does not degrade identically for the same amount of missing values. It depends on the attribute: For V303 (highest information gain attribute) we see an improvement in the classifier performance when increasing the number of missing values. For V189 this is not the case and we observe an inverse pattern.

3.3 Comparison with State of the Art

We compared our results to the scores on the challenge leaderboard of openml.org. The best run, when considering the absolute mean error, was achieved by executing a Support Vector Machine classifier (https://www.openml.org/r/6064623) with a radial basis function (rbl) kernel and complexity parameter of 1145.43. It achieved a mean absolute error of 0.0039, compared to 0.2225 by our test with WEKA's SMO (with PolyKernel, default tolerance parameter and complexity parameter = 15).

4 SUMMARY

In this assignment we used a dataset, containing sensor readings taken on a mobile phone of subjects doing particular movements, such as standing, lying, moving upstairs or moving downstairs. These readings were labeled with the particular class of type of movement. We used particularly two supervised algorithms for classification, observed how different parameters, sizes of training data and missing values influence the accuracy of results and the performance of the used algorithms. By tuning these algorithms we achieved high rates of correctly classified instances, such as 97.1356% by K-Nearest Neighbor algorithm or 98.7863% by Support Vector Machine algorithm. We also experimented with different strategies of dealing with missing values and show that in this dataset the replacement of missing values with the attribute mean of the respective class yields the highest classification accuracy.
The used dataset was well prepared as no preprocessing steps were needed. However, we realized that the size of our dataset should be decreased by sub sampling in order to ease the execution of algorithms and increase performance. We had issues in relation to replacing missing values for the entire dataset. As for our script, we used Java with maven to quickly start our code and get results out for the missing values on all six data sets.