# Pattern Recognition Coursework II


Orsika Bojtar
CID 01267809
ob116@ic.ac.uk

Peter Sarvari
CID 00987075
ps5714@ic.ac.uk

Imperial College London
S. Kensington, London SW7 2AZ



## Abstract

*In the following, we compare nearest neighbour (NN) classification accuracies for a given wine data set using different distance metrics. Secondly, we make use of a k-Means based technique (kMkNN) to improve the efficiency of NN. Lastly, we compare these results to that of Neural Networks.*


## 1. Q1 Distance Metrics

The given wine data contains 178 samples with 13 dimensions for each. First we tested how distance values between 3 pairs of points vary when different metrics [7] are used (see Figure 3 in Appendix). Then their influence on kNN classification accuracy (Figure 1) and performance (Figure 4 Appendix) have been examined. Even though the standard Euclidean distance is the closest to our general (mainly 1D/2D/3D) "distance" concept, it is clear from our results that other metrics perform better.

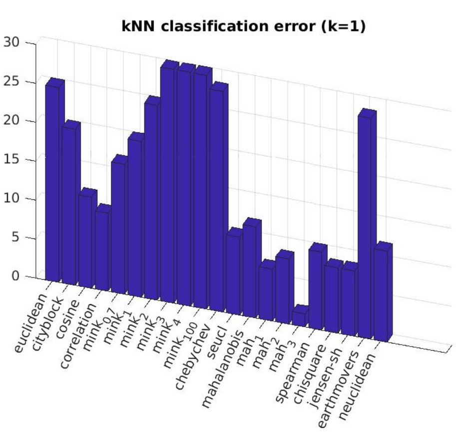


Figure 1. kNN accuracy for different distance metrics

In case of Euclidean/Minkowski distances, the largest features dominate the distance calculation, however those features may not be the most important for classification ([3] p66). One approach to get rid of this influence is to use normalized metrics (normalized Euclidean with unit length vectors/cosine metric, standardized Euclidean with samples normalized with the dimensional standard deviations). Another option is to use metrics that take into consideration the correlation between the data, again another is histogram-based calculation. The best performing metrics in our case were the Mahalanobis distance with 11.67% classification error (which was decreased to **1.67%** by tuning the covariance matrix, see later) Chi-Square and Jensen-Shannon(8.33%), correlation(10%), Spearman(10%), standardized Euclidean(10%), cosine(11.67%), and normalized Euclidean(11.67%) distances. For full results, see Table 2 in Appendix. This result corresponds to the experiment we previously ran on the actual distance values (see Figure 3 and Table 1 in Appendix), where Euclidean/Minkowski metrics resulted in large distance values for similar samples (data1,data2) quite commonly only because their original 'coordinates' were large. Contrary, metrics providing better classification accuracy reflected the underlying connections of the multidimensional data set.

### 1.1. Minkwoski distances

- $p = 1$ equals to Cityblock distance
- $p = 2$ equals to Euclidean distance
- $p = \inf$ is Chebychev distance (max value)

Minkowski with $p < 1$ is not a metric since the triangle inequality does not hold, though it has an interesting aspect weighing multidimensional changes over only one dimensional changes. Cityblock seems to perform better than Euclidean. For higher p values, the difference in the distances is negligible (see Table 1 in Appendix) so that kNN classification ends up using the same best data.

### 1.2. Relationship between $d_{nEuc}$ and $d_{Cos}$

Let us denote the length normalized vectors with $x_n = \frac{x}{\|x\|}$ and $y_n = \frac{y}{\|y\|}$ The Euclidean distance calculated on these unit-length vectors will be $d^2_{nEuc} = (x_n - y_n)^2 = x^2_n - 2x'_n y + y^2_n = 2 - 2x'_n y_n$, wheras the cosine metric gives us d$_{Cos} = 1 - \frac{x'y}{\|x\|\|y\|} = 1 - x'_n y_n$. The relationship between these two is determined by the equation

$$d_{nEuc} = \sqrt{2d_{Cos}} \quad or \quad d_{Cos} = \frac{d^2_{nEuc}}{2} \tag{1}$$

The kNN algorithm finds the smallest out of all pairwise distances. Since both square and square root functions are strictly monotonically increasing, the order of the pairwise distances will be the same in case of the two methods, resulting in the same (11.67%) classification error.

### 1.3. Mahalanobis distance

4 of our test metrics were Mahalanobis distances, but with different scaling matrices, A. A was chosen to be the covariance matrix of

- training samples $\rightarrow$ 11.67% classification error
- training samples belonging to class 1 $\rightarrow$ 6.67%
- training samples belonging to class 2 $\rightarrow$ 8.33%
- training samples belonging to class 3 $\rightarrow$ 1.67%

The confusion matrix for the nearest neighbour classification using Euclidean distance was

$$\begin{bmatrix} 19 & 1 & 0 \\ 3 & 14 & 3 \\ 3 & 5 & 12 \end{bmatrix} \tag{2}$$

We see that class 3 causes the most problems in terms of misclassification. So, we could try making it more separable. Indeed, the Mahalanobis distance using covariance matrix specific to class 3 results in the best nearest neighbour prediction (98.33% accuracy). Interestingly, however, the same procedure using class 1 covariance resulted in better prediction than using class 2 covariance. All methods using class-specific covariance matrices resulted in better prediction than using the whole covariance matrix.

### 1.4. Histogram distances

The bin-by-bin Jhensen-Shannon and Chi-Square metrics are two of our best methods. Their advantage lies in the fact that they weigh dimensions (bins) differently, reducing the effect of large bins. Since they do not connect multiple bins, they can be safely used on non-histogram data. However, using Earth Mover's distance (which does connect multiple bins by taking into account their relative location) on the wine data set is not recommended. This is because the 13 features correspond to different concepts (*e.g.*'Alcohol' and 'Color'). Hence, it does not make sense to say, for example, that moving a bit of "earth" from the 'Alcohol' feature to the 'Total phenols' feature should result in six times the work than moving the same amount of "earth" from the 'Magnesium' feature to the 'Total phenols' feature and for this reason Earth Mover's distance is not really applicable here. Not to mention that our data would still be valid corresponding to the same physical meaning if we interchanged the order of the features, however Earth Mover's distance would be completely different. Nevertheless, we implemented the Earth Mover's distance in Matlab and tested it on our dataset, and - as we expected - its performance was one of the worst.

## 2. Q2 K-means clustering

One clever way of using kmeans to reduce the complexity of the nearest neighbour algorithm is discussed in [8]. The so-called kMkNN algorithm uses the triangle inequality to save the calculation of training points that are far from the given testing point. We implemented this algorithm in Matlab for the different distance metrics and verified that unnecessary distance calculations are skipped, and the resulting classification error is the same for all the metrics for which the triangle inequality applies (see Figure 2).

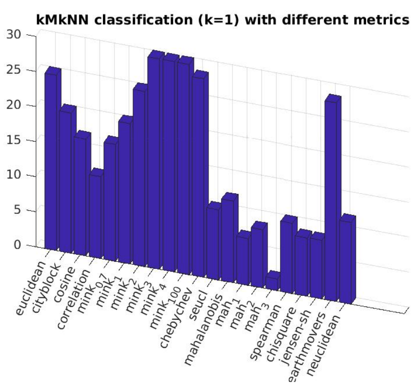


Figure 2. kMkNN accuracy for different distance metrics

For metrics where the triangle inequality does not apply (*e.g.* cosine, Mikowski with p = 0.7 and correlation metrics), kMkNN might give worst results for classification. Indeed, for cosine and correlation distance, we get bigger classification error. Note that even if the metric satisfies the triangle inequality, small differences can still arise in cases when the distance between the current testing point and multiple training points are the same and minimal. In this case the sample that is found first will determine the class of the test sample for both kMkNN and the Matlab built-in kNN. We also compared the runtime of the two algorithms. Since the sample size as well as the dimension is small, we did not observe reduction in run-time, in fact, kMkNN took longer to run. This is slightly surprising (we only measured the runtime for the search stage (as in [8])), but considering the fact that built-in Matlab functions are highly optimized for runtime, the tiny sample/dimension size and the additional overhead caused by the distance calculations between the testing points and cluster centres as well as their sorting, the results are understandable. Also, note that in the original paper runtime also increased for certain small datasets ([8] Table I & III). When one runs the code, they can immediately see all the distance calculations that have been skipped (uncomment corresponding lines 189-192, source code in Appendix) confirming that the method indeed reduces the theoretical complexity of the nearest neighbour classifier, whilst providing an exact result for proper distance metrics, whose properties include the triangular inequality.

## 3. Q3 Neural Network

We used Matlabs Neural Network Toolbox to investigate the classification problem. The toolbox automatically divides the inputted dataset randomly into train, validation and test sets, however we were explicitly told to use 2 sets and the division was also given in the coursework. Hence, rather than using the graphical interface, we generated the script and manually gave Matlab the training and test set. Then, we kept changing the settings to find the best combination for test set accuracy. (Note that this result indeed reflects the best parameters for the given data division, but the resultant (98.33%) accuracy is not what we would expect on an unseen dataset this is because we used the test set to tune the hyperparameters.) We tried normalization and different transfer (activation) functions, network layouts and optimization methods:

- Data normalized / unnormalized
  Data normalization is standard practice in Machine Learning to reduce the mean of the training input data to zero and its standard deviation to 1.
- Sigmoid or Tanh (default) activation function (Matlab calls them logsig and tansig, respectively)
  In practice, tanh nonlinearity is always preferred to sigmoid ([2] p17), however we wanted to see whether this usual practical choice is supported by the wine example.
- Optimization methods [6]
  - Scaled conjugate gradient backpropagation (default)
  - Conjugate gradient backpropagation with Polak-Ribiere updates
  - BFGS quasi-Newton backpropagation
  - RPROP backpropagation
  - Gradient descent with momentum and adaptive learning rate backpropagation

  The reasons for choosing the 5 optimization methods can be summarized as follows: the first 3 were in detail discussed in the EE4-29, Optimization Course Notes[1], the advantages of adaptive learning rate and momentum are outlined in the CS231n course [4] and RPROP was the optimization method used for neural networks in the Machine Learning and Neural Computation (BE9-MMLNC) Course by the Bioengineering Dept.
- Network architecture
  - 1 hidden layer with 10 or 8 or 6 or 4 neurons (4 options)
  - 2 hidden layers with 10 & 8 or 10 & 6 or 10 & 4 or 8 & 6 or 8 & 4 or 6 & 4 neurons (6 options)
  - 3 hidden layers with 10 & 8 & 6 or 10 & 8 & 4 or 10 & 6 & 4 neurons (3 options)
  - 4 hidden layers with 10 & 8 & 6 & 4 neurons (1 option)

  The reason for such network architecture is as follows: we start off with 13 neurons in the first layer (corresponding to the number of features) and we try to reduce them to 3 neurons in the last layer (they give the probabilities that the sample belongs to each of the 3 class). Trying all possibilities would take a lot of time, so we needed to make reasonable assumptions to sample the possible options in a good way. We assumed that the reduction in number of neurons from the first layer to the second is at least 3, and from all other layers to the next, at least 2, except for last two layers, where the reduction can be 1 (e.g. 4 to 3).

All other settings are left as default [5]. The best result (98.33% accuracy) was achieved for 3 hidden layers with 10, 8 and 6 neurons in them with the Polak-Ribiere optimization method and tanh activation function (as expected). The same result was achieved for the non-normalized and the normalized data. For full results see the Appendix. Note that interestingly, the best accuracy with the Neural Network algorithm is the same as the best accuracy achieved using the nearest neighbour algorithm (see Q1). Additionally it can be noted about regularization that Matlab avoids overfitting of the neural network via the early stopping method. Validation cost (cross-entropy cost) is constantly monitored, and (by default) if it increases for more than 6 epochs (model starts overfitting) the training is stopped and the parameters corresponding to the best achieved validation cost are returned.

## References


[1] A. Astolfi. Optimization. an introduction. *Imperial College London*.

[2] K. Mikolajczyk. Prdeeplearning slides.

[3] K. Mikolajczyk. Prlecture$_distance_metricslides$.

[4] http://cs231n.github.io/neural-networks-3/. Cs231n convolutional neural networks for visual recognition. Accessed: 2017-12-18.

[5] https://uk.mathworks.com/help/nnet/ref/trainingoptions.html. Matlab options for training neural network. Accessed: 2017-12-18.

[6] https://uk.mathworks.com/help/nnet/ug/train-and-apply-multilayer-neural-networks.html. Matlab train and apply multilayer neural networks. Accessed: 2017-12-18.

[7] https://uk.mathworks.com/help/stats/pdist2.html?searchHighlight=pdist2&s_tid=doc_srchtitle. Matlab distance functions. Accessed: 2017-12-18.

[8] X. Wang. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. *IJCNN*, pages 1293–1299, 2011.


## Appendix

| Metric | dist(data1, data2) | dist(data1, data3) | dist(data1, data100) |
|---|---|---|---|
| 'euclidean' | 245,112465819264 | 90,0580196317907 | 577,018849432148 |
| 'cityblock' | 256,590000000000 | 95,0900000000000 | 586,860000000000 |
| 'cosine' | 0,000774391831877552 | 1,38629252954825e-05 | 0,00858437897709152 |
| 'correlation' | 0,000724189778435247 | 1,19112600672766e-05 | 0,00829612787438550 |
| 'mink_{0.7}' | 313,562593419051 | 118,825971245568 | 665,063214230027 |
| 'mink_1' | 256,590000000000 | 95,0900000000000 | 586,860000000000 |
| 'mink_2' | 245,112465819264 | 90,0580196317907 | 577,018849432148 |
| 'mink_3' | 245,001968555417 | 90,0011709375639 | 577,000062095193 |
| 'mink_4' | 245,000041224554 | 90,0000283220812 | 577,000000244786 |
| 'mink_{100}' | 245,000000000000 | 90,0000000000000 | 577,000000000000 |
| 'chebychev' | 245 | 90 | 577 |
| 'seucl' | 2,64167021695041 | 1,70983596412489 | 2,95276169822763 |
| 'mahalanobis' | 3,20671261142587 | 3,04362233330445 | 3,14469539419700 |
| 'mah_1' | 4,40990982734658 | 3,96601639717757 | 4,94778432408051 |
| 'mah_2' | 4,21281419211033 | 3,38660827672682 | 5,81715992773014 |
| 'mah_3' | 5,42447479540173 | 6,62805821012773 | 10,6119671365375 |
| 'spearman' | 0,0563952558845489 | 0,00549450549450514 | 0,0164835164835162 |
| 'chisquare' | 4,17825601302647 | 1,43019708586298 | 10,7217169286409 |
| 'jensen-sh' | 2,96011084052615 | 1,01158664733248 | 7,68685373001280 |
| 'earthmovers' | 0,355863749745883 | 0,0516163179402210 | 1,07076313383277 |
| 'neuclidean' | 0,0393545888525780 | 0,00526553421705857 | 0,131029607166409 |

Table 1. Pairwise distances (see on Figure 3)

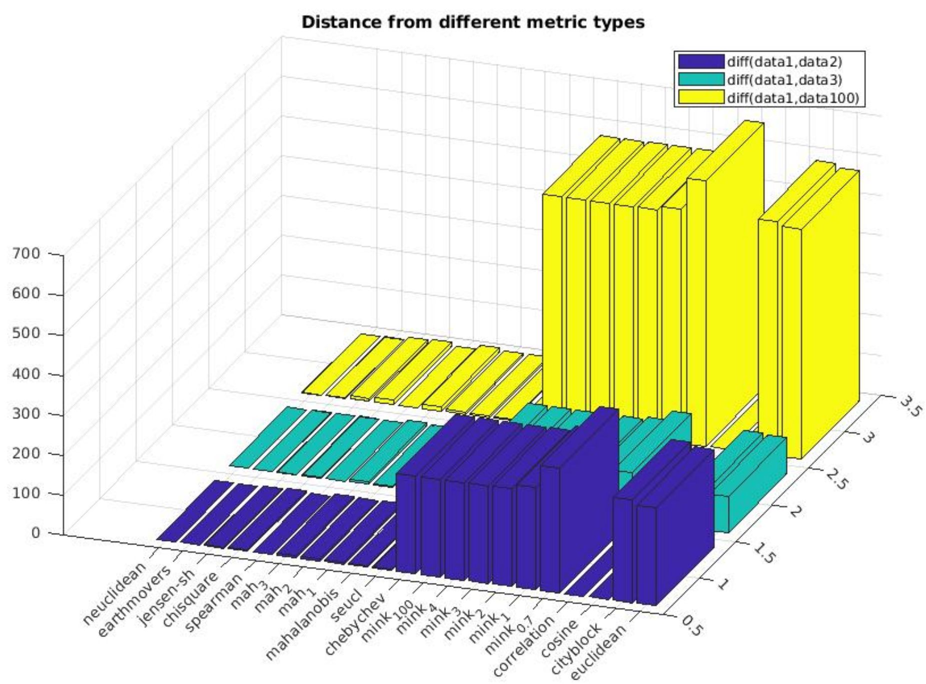


Figure 3. 3 different pairwise distances calculated by different distance metrics. Blue is between wine data 1 and data 2 (both belong to class 1), green is between wine data 1 and 3 (both belong to class 1), yellow is between dissimilar samples wine data 1 and wine data 100 (class 1 & 2 respectively). Reasonably the difference calculated for dissimilar samples are generally bigger (see Table 1).

| Metric | kNN cl. error [%] | kNN time [s] | kMkNN cl. error [%] | kMkNN time [s] |
|---|---|---|---|---|
| 'euclidean' | 25 | 0,013268 | 25 | 0,107013 |
| 'cityblock' | 20 | 0,013773 | 20 | 0,114482 |
| **'cosine'** | 11.67 | 0,014844 | 16,6767 | 0,159208 |
| **'correlation'** | 10 | 0,01622 | 11,67 | 0,195208 |
| 'mink_{0.7}' | 16.67 | 0,022196 | 16,67 | 0,137641 |
| 'mink_1' | 20 | 0,016773 | 20 | 0,116791 |
| 'mink_2' | 25 | 0,011528 | 25 | 0,09529 |
| 'mink_3' | 30 | 0,0173 | 30 | 0,094472 |
| 'mink_4' | 30 | 0,017058 | 30 | 0,096684 |
| 'mink_{100}' | 30 | 0,01647 | 30 | 0,116865 |
| 'chebychev' | 28.33 | 0,009712 | 28,33 | 0,085302 |
| **'seucl'** | 10 | 0,013131 | 10 | 0,422215 |
| **'mahalanobis'** | 11.67 | 0,015216 | 11,67 | 1,091037 |
| **'mah_1'** | 6.67 | 0,013335 | 6,67 | 0,529778 |
| **'mah_2'** | 8.33 | 0,012966 | 8,333 | 0,424762 |
| **'mah_3'** | 1.67 | 0,014324 | 1,67 | 0,432434 |
| **'spearman'** | 10 | 0,019294 | 10 | 0,539413 |
| **'chisquare'** | 8.33 | 0,016756 | 8,33 | 0,265906 |
| **'jensen-sh'** | 8.33 | 0,021262 | 8,33 | 0,282535 |
| 'earthmovers' | 28.33 | 0,015272 | 28,33 | 0,131636 |
| **'neuclidean'** | 11.67 | 0,010679 | 11,67 | 0,265922 |

Table 2. kNN and kMkNN efficiency using different metrics

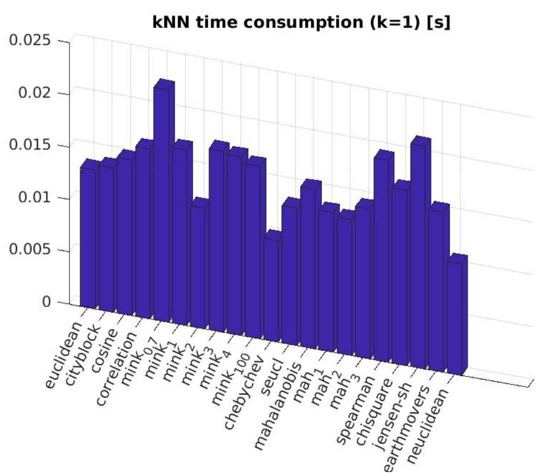


Figure 4. kNN speed for different distance metrics

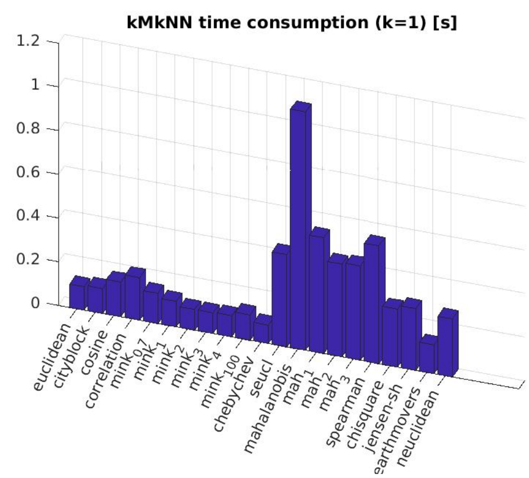


Figure 5. kMkNN speed for different distance metrics

| Optimization | Scaled cgb | Cgb w Polak-Ribier | BFGS | RPROP | Grad. d w m&alrb |
|---|---|---|---|---|---|
| [10] | 0,933333333333333 | 0,966666666666667 | 0,633333333333333 | 0,933333333333333 | 0,950000000000000 |
| [8] | 0,933333333333333 | 0,716666666666667 | 0,916666666666667 | 0,933333333333333 | 0,933333333333333 |
| [6] | 0,716666666666667 | 0,950000000000000 | 0,966666666666667 | 0,966666666666667 | 0,933333333333333 |
| [4] | 0,616666666666667 | 0,616666666666667 | 0,616666666666667 | 0,950000000000000 | 0,966666666666667 |
| [10 8] | 0,916666666666667 | 0,933333333333333 | 0,650000000000000 | 0,950000000000000 | 0,933333333333333 |
| [10 6] | 0,850000000000000 | 0,633333333333333 | 0,633333333333333 | 0,866666666666667 | 0,866666666666667 |
| [10 4] | 0,866666666666667 | 0,900000000000000 | 0,633333333333333 | 0,900000000000000 | 0,966666666666667 |
| [8 6] | 0,933333333333333 | 0,650000000000000 | 0,650000000000000 | 0,900000000000000 | 0,650000000000000 |
| [8 4] | 0,950000000000000 | 0,966666666666667 | 0,633333333333333 | 0,866666666666667 | 0,633333333333333 |
| [6 4] | 0,900000000000000 | 0,900000000000000 | 0,650000000000000 | 0,916666666666667 | 0,900000000000000 |
| [10 8 6] | 0,333333333333333 | **0,983333333333333** | 0,333333333333333 | 0,966666666666667 | 0,916666666666667 |
| [10 8 4] | 0,850000000000000 | 0,633333333333333 | 0,616666666666667 | 0,866666666666667 | 0,883333333333333 |
| [10,6 4] | 0,900000000000000 | 0,633333333333333 | 0,633333333333333 | 0,900000000000000 | 0,916666666666667 |
| [10 8 6 4] | 0,616666666666667 | 0,633333333333333 | 0,633333333333333 | 0,850000000000000 | 0,583333333333333 |

Table 3. Accuracy of Neural Network with tanh activation function
(unnormalized data)

| Optimization | Scaled cgb. | Cgb w Polak-Ribier | BFGS | RPROP | Grad. d w m&alrb |
|---|---|---|---|---|---|
| [10] | 0,933333333333333 | 0,966666666666667 | 0,633333333333333 | 0,933333333333333 | 0,950000000000000 |
| [8] | 0,933333333333333 | 0,716666666666667 | 0,916666666666667 | 0,933333333333333 | 0,933333333333333 |
| [6] | 0,716666666666667 | 0,950000000000000 | 0,966666666666667 | 0,966666666666667 | 0,933333333333333 |
| [4] | 0,616666666666667 | 0,616666666666667 | 0,616666666666667 | 0,950000000000000 | 0,966666666666667 |
| [10 8] | 0,916666666666667 | 0,933333333333333 | 0,650000000000000 | 0,950000000000000 | 0,933333333333333 |
| [10 6] | 0,850000000000000 | 0,633333333333333 | 0,633333333333333 | 0,866666666666667 | 0,866666666666667 |
| [10 4] | 0,866666666666667 | 0,900000000000000 | 0,633333333333333 | 0,900000000000000 | 0,966666666666667 |
| [8 6] | 0,933333333333333 | 0,650000000000000 | 0,650000000000000 | 0,900000000000000 | 0,650000000000000 |
| [8 4] | 0,950000000000000 | 0,966666666666667 | 0,633333333333333 | 0,866666666666667 | 0,633333333333333 |
| [6 4] | 0,900000000000000 | 0,900000000000000 | 0,650000000000000 | 0,916666666666667 | 0,900000000000000 |
| [10 8 6] | 0,333333333333333 | **0,983333333333333** | 0,333333333333333 | 0,966666666666667 | 0,916666666666667 |
| [10 8 4] | 0,850000000000000 | 0,633333333333333 | 0,616666666666667 | 0,866666666666667 | 0,883333333333333 |
| [10,6 4] | 0,900000000000000 | 0,633333333333333 | 0,633333333333333 | 0,900000000000000 | 0,916666666666667 |
| [10 8 6 4] | 0,616666666666667 | 0,633333333333333 | 0,633333333333333 | 0,850000000000000 | 0,583333333333333 |

Table 4. Accuracy of Neural Network with tanh activation function
(normalized data)

| Optimization | Scaled cgb. | Cgb w Polak-Ribier | BFGS | RPROP | Grad. d w m&alrb |
|---|---|---|---|---|---|
| [10] | 0,350000000000000 | 0,433333333333333 | 0,333333333333333 | 0,883333333333333 | 0,316666666666667 |
| [8] | 0,850000000000000 | 0,333333333333333 | 0,333333333333333 | 0,700000000000000 | 0,650000000000000 |
| [6] | 0,533333333333333 | 0,333333333333333 | 0,333333333333333 | 0,350000000000000 | 0,566666666666667 |
| [4] | 0,900000000000000 | 0,350000000000000 | 0,333333333333333 | 0,483333333333333 | 0,750000000000000 |
| [10 8] | 0,400000000000000 | 0,333333333333333 | 0,333333333333333 | 0,800000000000000 | 0,633333333333333 |
| [10 6] | 0,733333333333333 | 0,333333333333333 | 0,333333333333333 | 0,550000000000000 | 0,666666666666667 |
| [10 4] | 0,700000000000000 | 0,333333333333333 | 0,333333333333333 | 0,883333333333333 | 0,800000000000000 |
| [8 6] | 0,633333333333333 | 0,333333333333333 | 0,333333333333333 | 0,483333333333333 | 0,266666666666667 |
| [8 4] | 0,616666666666667 | 0,333333333333333 | 0,333333333333333 | 0,483333333333333 | 0,616666666666667 |
| [6 4] | 0,566666666666667 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,466666666666667 |
| [10 8 6] | 0,583333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 |
| [10 8 4] | 0,766666666666667 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,516666666666667 |
| [10,6 4] | 0,600000000000000 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,600000000000000 |
| [10 8 6 4] | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 |

Table 5. Accuracy of Neural Network with sigmoid activation function (unnormalized data)

| Optimization | Scaled cgb. | Cgb w Polak-Ribier | BFGS | RPROP | Grad. d w m&alrb |
|---|---|---|---|---|---|
| [10] | 0,350000000000000 | 0,433333333333333 | 0,333333333333333 | 0,883333333333333 | 0,316666666666667 |
| [8] | 0,850000000000000 | 0,333333333333333 | 0,333333333333333 | 0,700000000000000 | 0,650000000000000 |
| [6] | 0,533333333333333 | 0,333333333333333 | 0,333333333333333 | 0,350000000000000 | 0,566666666666667 |
| [4] | 0,900000000000000 | 0,350000000000000 | 0,333333333333333 | 0,483333333333333 | 0,750000000000000 |
| [10 8] | 0,400000000000000 | 0,333333333333333 | 0,333333333333333 | 0,800000000000000 | 0,633333333333333 |
| [10 6] | 0,733333333333333 | 0,333333333333333 | 0,333333333333333 | 0,550000000000000 | 0,666666666666667 |
| [10 4] | 0,700000000000000 | 0,333333333333333 | 0,333333333333333 | 0,883333333333333 | 0,800000000000000 |
| [8 6] | 0,633333333333333 | 0,333333333333333 | 0,333333333333333 | 0,483333333333333 | 0,266666666666667 |
| [8 4] | 0,616666666666667 | 0,333333333333333 | 0,333333333333333 | 0,483333333333333 | 0,616666666666667 |
| [6 4] | 0,566666666666667 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,466666666666667 |
| [10 8 6] | 0,583333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 |
| [10 8 4] | 0,766666666666667 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,516666666666667 |
| [10,6 4] | 0,600000000000000 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,600000000000000 |
| [10 8 6 4] | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 | 0,333333333333333 |

Table 6. Accuracy of Neural Network with sigmoid activation function (normalized data)

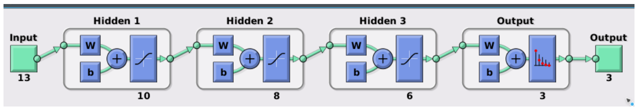


Figure 6. Layer architecture for the best Neural Network scenario ([10 8 6] with Polak Ribier, tanh)

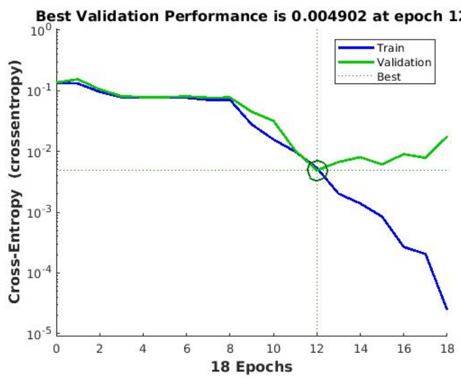


Figure 7. Performance for the best Neural Network scenario ([10 8 6] with Polak Ribier, tanh)

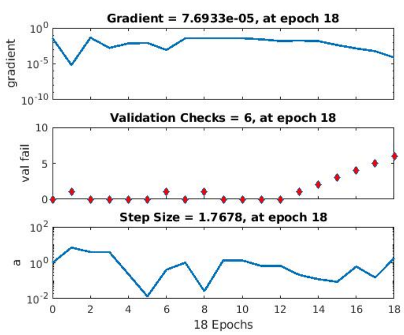


Figure 8. Training state for the best Neural Network scenario ([10 8 6] with Polak Ribier, tanh)

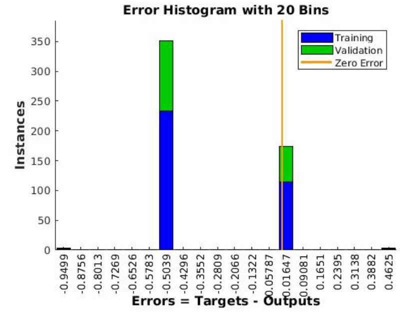


Figure 9. Error histogram for the best Neural Network scenario ([10 8 6] with Polak Ribier, tanh)


Figure 10. Confusion matrices for the best Neural Network scenario ([10 8 6] with Polak Ribier, tanh). Since the test data functioned as our validation set, only the first two confusion matrices provide reasonable information.

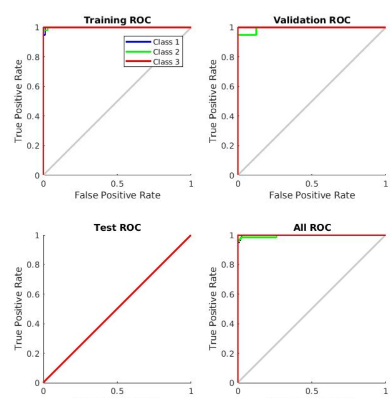


Figure 11. Receiver Operating Characteristic for the best Neural Network scenario ([10 8 6] with Polak Ribier, tanh)

## 4. Source Code

### 4.1. metrics_final.m

```matlab
close all;
clear;
clc

%% Read data and define metrics

wine = dlmread('wine.data.csv');
numberofdata = size(wine,1); %178
dimensions = size(wine,2); %15 (1+1+13features)
classes = 3;
rng(7);

metric_types = {'euclidean', 'cityblock', 'cosine', 'correlation', 'mink_{0.7}', 'mink_1', 'mink_2',...
                'mink_3', 'mink_4', 'mink_{100}', 'chebychev','seucl', ...
                'mahalanobis', 'mah_1', 'mah_2', 'mah_3', 'spearman', ...
                'chisquare', 'jensen-sh', 'earthmovers', 'neuclidean'};

metric_number = size(metric_types,2);

%% Split to training & testing data
wine_training = wine(wine(:,1)==1,:);
wine_testing = wine(wine(:,1)==2,:);

meansub = wine_training(:,3:end)-mean(wine_training(:,3:end),1);
covarmat = 1/size(wine_training, 1)*(meansub'*meansub);
class1 = wine_training(wine_training(:,2)==1,3:end)-mean(wine_training(wine_training(:,2)==1,3:end),1);
class2 = wine_training(wine_training(:,2)==2,3:end)-mean(wine_training(wine_training(:,2)==2,3:end),1);
class3 = wine_training(wine_training(:,2)==3,3:end)-mean(wine_training(wine_training(:,2)==3,3:end),1);
covarmat_class1 = 1/size(class1,1) * (class1'*class1);
covarmat_class2 = 1/size(class2,1) * (class2'*class2);
covarmat_class3 = 1/size(class3,1) * (class3'*class3);
standard_dev_vec = std(wine_training(:,3:end), 1);

%% Visualize difference between distance metrics
differences = zeros(numberofdata, numberofdata, metric_number);
for i=1:metric_number
    [distance, dpar] = getMetricType_final(i, metric_types, covarmat, standard_dev_vec, covarmat_class1, covarmat_class2, covarmat_class3);
    X = wine;
    if strcmp(distance, 'minkowski') || strcmp(distance, 'mahalanobis') || strcmp(distance, 'seuclidean')
        Z = squareform(pdist(X(:,3:end),distance,dpar));
    elseif strcmp(distance, 'neuclidean')
        Z = squareform(pdist(normr(X(:,3:end)),'euclidean'));
    else
        Z = squareform(pdist(X(:,3:end),distance));
```

```
    end
    differences(:,:,i) = Z;
end
figure
hold on
b1 = reshape(differences(1,2,:),1,size(differences,3));
b2 = reshape(differences(1,3,:),1,size(differences,3));
b100 = reshape(differences(1,100,:),1,size(differences,3));
b = bar3(1:metric_number,[b100',b2',b1']);
title('Distance from different metric types');
set(gca,'yticklabel',metric_types,'YTick',1:numel(metric_types));
ax = gca;
ax.YTickLabelRotation = 45;
ax.XTickLabelRotation = -45;
grid on
l = cell(1,3);
l{1}='diff(data1,data2)'; l{2}='diff(data1,data3)'; l{3}='diff(data1,data100)';
legend(b,l);
view(-70,40);


%% Distance metrics / NN-classification (k=1 based on lectures)
temp = zeros(metric_number,4);
time_knn = zeros(1,metric_number);
pred_maha_saved = -1*ones(size(wine_testing,1),4);
for i=1:metric_number
    [distance, dpar] = getMetricType_final(i, metric_types, covarmat,
        standard_dev_vec, covarmat_class1, covarmat_class2, covarmat_class3);
    tic
    [temp(i,3), ~, pred1] = error_calc2(wine_training, wine_testing, 1, distance,
        dpar);
    time_knn(i) = toc;

    if isequal(dpar, covarmat)
        pred_maha_saved(:,1) = pred1;
    elseif isequal(dpar, covarmat_class1)
        pred_maha_saved(:,2) = pred1;
    elseif isequal(dpar, covarmat_class2)
        pred_maha_saved(:,3) = pred1;
    elseif isequal(dpar, covarmat_class3)
        pred_maha_saved(:,4) = pred1;
    end
end

%Interesting, but nothing intuitive, not sure if we should include this
%result
class1_acc = sum(pred_maha_saved(wine_testing(:,2)==1, :) == 1);
class2_acc = sum(pred_maha_saved(wine_testing(:,2)==2, :) == 2);
class3_acc = sum(pred_maha_saved(wine_testing(:,2)==3, :) == 3);


figure
bar3(100*temp(:,3)./size(wine_testing,1)); %plot error rate %
```

```
grid on
title('kNN classification error (k=1)');
set(gca,'Yticklabel',metric_types,'YTick',1:numel(metric_types));
ax = gca;
ax.YTickLabelRotation = 65;
view(-70,30);
saveas(gcf,['knn_acc.jpg']);

figure
bar3(time_knn);
grid on
title('kNN time consumption (k=1) [s]');
set(gca,'Yticklabel',metric_types,'YTick',1:numel(metric_types));
ax = gca;
ax.YTickLabelRotation = 65;
view(-70,30);
saveas(gcf,['knn_time.jpg']);

%% Earth-movers distance
% How it changes if we shuffle the order of the features
[distance, dpar] = getMetricType_final(getIndex('earthmovers', metric_types), metric_types, covarmat, standard_dev_vec, covarmat_class1, covarmat_class2, covarmat_class3);
eclass = zeros(1,40);
[eclass(1), ~, pred1] = error_calc2(wine_training, wine_testing, 1, distance, dpar);

wine_end = wine_training(:,3:end);
wine_training_shuffledbins = zeros(size(wine_training));
oldorder = 1:13;
for i=2:40
    order = randperm(size(wine_end,2));
    while nnz(order - oldorder) == 0
        i;
            order = randperm(size(wine_end,2));
    end
    wine_training_shuffledbins = [wine_training(:,1:2) wine_end(:, order) ];
    [eclass(i), ~, pred2] = error_calc2(wine_training_shuffledbins, wine_testing, 1, distance, dpar);
    oldorder = order;
end

figure
bar(100*eclass./size(wine_testing,1));
grid on
title('Earth movers distance (shuffled bins)');
ylabel('Classification error (%)');

%% Exact kMkNN search

%Build-up stage
rng(7);
kc = floor(sqrt(size(wine_training,1)));
```

```
tt = randperm(size(wine_training,1));
x = tt(1:kc);
initial_means = wine_training(x,:);
pi = cell(kc);
di = cell(kc);
maxdistance = 10^5;
k = 1;
pred = zeros(size(wine_testing,1),1);
speed = zeros(1,metric_number);
error = zeros(1,metric_number);
for mn=1:metric_number
    [distance, dpar] = getMetricType_final(mn, metric_types, covarmat, standard_dev_vec, covarmat_class1, covarmat_class2, covarmat_class3);
    [class_kmknn_vec, mean_kmknn_vec] =  simplekmeans(wine_training(:,3:end), initial_means(:,3:end),100, distance, dpar);
    for class=1:kc
        temp = wine_training(class_kmknn_vec==class,:);
        if strcmp(distance, 'minkowski') || strcmp(distance, 'mahalanobis') || strcmp(distance, 'seuclidean')
            di_temp = pdist2(temp(:,3:end), mean_kmknn_vec(class,:), distance, dpar);
        elseif strcmp(distance, 'neuclidean')
            di_temp = pdist2(normr(temp(:,3:end)), normr(mean_kmknn_vec(class,:)), 'euclidean');
        else
            di_temp = pdist2(temp(:,3:end), mean_kmknn_vec(class,:), distance);
        end
        [di_temp,index] = sort(di_temp, 'descend');
        di{class} = di_temp;
        pi{class} = temp(index,:);
    end
%Search stage
    tic
    for ts = 1:size(wine_testing,1)
        object = maxdistance*ones(k,2); %first col is actual distance second is the class
        test_sample = wine_testing(ts,3:end);
        if strcmp(distance, 'minkowski') || strcmp(distance, 'mahalanobis') || strcmp(distance, 'seuclidean')
            dist_to_centres = pdist2(test_sample, mean_kmknn_vec, distance, dpar);
        elseif strcmp(distance, 'neuclidean')
            dist_to_centres = pdist2(normr(test_sample), normr(mean_kmknn_vec), 'euclidean');
        else
            dist_to_centres = pdist2(test_sample, mean_kmknn_vec, distance);
        end
        [~, ix] = sort(dist_to_centres, 'ascend');
        for cluster = ix
            for member = 1:length(di{cluster})
                pc = di{cluster}(member);
                compare = dist_to_centres(cluster)-pc; %If abs taken, does not work!
                if max(object(:,1)) <= compare
```

```
%                           disp('Some training samples skipped due to triangular conidition')
%                           disp(['ts:', num2str(ts)])
%                           disp(['cluster:', num2str(cluster)])
%                           disp(['member:', num2str(member)])
                        break
                    else
                        if strcmp(distance, 'minkowski') || strcmp(distance, 'mahalanobis') || strcmp(distance, 'seuclidean')
                            temp = pdist2(test_sample, pi{cluster}(member,3:end), distance, dpar);
                        elseif strcmp(distance, 'neuclidean')
                            temp = pdist2(normr(test_sample), normr(pi{cluster}(member,3:end)), 'euclidean');
                        else
                            temp = pdist2(test_sample, pi{cluster}(member,3:end), distance);
                        end
                        if temp < max(object(:,1))
                            [~,idx] = max(object(:,1));
                            object(idx,:) = [temp, pi{cluster}(member,2)];
                        end
                    end
                end
            end
            pred(ts) = mode(object(:,2));
            % This is to be used is k>1 and majority voting (mode) cannot decide
            %       [~, ~, C] = mode(object(:,2));
            %       C = cell2mat(C);
            %       MV_dist = zeros(1,length(c)); %distances for majority voted classes
            %       for j = 1:length(C)
            %           class = C(j);
            %           MV_dist(j) = mean(object(object(:,2) == class, 1));
            %       end
            %       [~, pred(ts)] = min(MV_dist);
        end
        speed(mn) = toc;
        error(mn) = nnz(pred-wine_testing(:,2));
end
figure
bar3(100*error./size(wine_testing,1)); %plot error rate %
grid on
title('kMkNN classification (k=1) with different metrics');
set(gca,'Yticklabel',metric_types,'YTick',1:numel(metric_types));
%ylabel('Classification error (%)');
ax = gca;
ax.YTickLabelRotation = 65;
view(-70,30);
%saveas(gcf,['kmknn_acc.jpg']);
figure
bar3(speed);
title('kMkNN time consumption (k=1) [s]');
%set(gca,'xticklabel',metric_types,'XTick',1:numel(metric_types));
```

```
%ylabel('Time consumption [s]');
set(gca,'Yticklabel',metric_types,'YTick',1:numel(metric_types));
ax = gca;
ax.YTickLabelRotation = 65;
view(-70,30);
saveas(gcf,['kmknn_time.jpg']);
```

### 4.2. error_calc2.m

```
function [n_testing_err, n_testing_succ, classif_for_testing] = error_calc2(wine_training, wine_testing, k, distance, dpar)
%dim 1: tr/te, 2: class, 3-15: real features
%Try classification
if strcmp(distance, 'neuclidean')
    mdl = fitcknn(normr(wine_training(:,3:end)), wine_training(:,2), 'NumNeighbors',k,'Distance', 'euclidean');
else
    mdl = fitcknn(wine_training(:,3:end), wine_training(:,2), 'NumNeighbors',k,'Distance', distance);
end
if strcmp(distance, 'minkowski') || strcmp(distance, 'mahalanobis') || strcmp(distance, 'seuclidean')
    mdl.DistParameter = dpar;
end
% classif_for_training = predict(mdl,wine_training(:,3:end));
if strcmp(distance, 'neuclidean')
    classif_for_testing = predict(mdl,normr(wine_testing(:,3:end)));
else
    classif_for_testing = predict(mdl,wine_testing(:,3:end));
end

%Error for training data (should be zero)
% error_for_training = classif_for_training - wine_training(:,2);
% n_training_err = nnz(error_for_training); %number of failures
% n_training_succ = size(error_for_training,1) - n_training_err; %number of successes
%Error for testing data (interesting to rate effectiveness)
error_for_testing = classif_for_testing - wine_testing(:,2);
n_testing_err = nnz(error_for_testing); %number of failures
n_testing_succ = size(error_for_testing,1) - n_testing_err; %number of successes

end
```

### 4.3. simplekmeans.m

```
function [class_vec, mean_vec] = simplekmeans(vec, initial_means, maxiterkmeans, distance, dpar)
%Class 1 is first row vector in initial_means

mean_vec = initial_means;
mean_minus = zeros(size(initial_means));
index = 0;
K = size(initial_means, 1); %2 in test data, 3 in wine
while ~isequal(mean_minus, mean_vec)
    mean_minus = mean_vec;
%     res_matrix = bsxfun(@plus, sum(vec.^2, 2), bsxfun(@minus, (sum(mean_vec.^2, 2))', 2*vec*mean_vec')); %I want (SAMPLEi - MUx)^2, I calculate SAMPLEi^2+MUx^2-2*SAMPLEi*MUx
    %X = [vec; mean_vec]; Use pdist2, more efficient, no unneccessary
    %distance calcs!
    if strcmp(distance, 'minkowski') || strcmp(distance, 'mahalanobis') || strcmp(distance, 'seuclidean')
        res_matrix = pdist2(vec,initial_means,distance,dpar);
    elseif strcmp(distance, 'neuclidean')
        res_matrix = pdist2(normr(vec), normr(initial_means), 'euclidean');
    else
        res_matrix = pdist2(vec,initial_means,distance);
    end
%     Z(1:size(X,1)-K,end-K+1:end);
%     res_matrix = Z(1:size(X,1)-K,end-K+1:end);

    [~, ix] = min(res_matrix, [], 2);
    class_vec = ix;
    for class = 1:K
        mean_vec(class,:) = mean(vec(class_vec==class, :), 1);
    end
    index = index + 1;
    if index > maxiterkmeans
        break
    end
    %[mean_minus mean_vec] %NaN was caused by centroids being the same
    %because multiple same entries in original dataset
    %pause
end
%index;
end

% To check exercise results in 2010 exam
%
% init_mean = [1 0 0; 0 1 0]
% vec = [1 3 4; 4 2 5; 2 5 6; 1 3 3; 2 0 4; 4 1 3]
% [class_vec, mean_vec] = simplekmeans(vec, init_mean)
```

### 4.4. getMetricType_final.m

```
function [ distance , dpar ] = getMetricType_final( i , metric_type , covarmat ,
    standard_dev_vec , covarmatclass1 , covarmatclass2 , covarmatclass3)

dpar = 0;
    switch i
        case getIndex('euclidean', metric_type)
            distance = 'euclidean';
        case getIndex('cityblock', metric_type)
            distance = 'cityblock';
        case getIndex('cosine', metric_type)
            distance = 'cosine';
        case getIndex('correlation', metric_type)
            distance = 'correlation';
        case getIndex('neuclidean', metric_type)
            distance = 'neuclidean';
        case getIndex('crosscorr', metric_type)
            crosscorr = @(x,Z) (x*Z')';
            distance = crosscorr;
        case getIndex('mink_{0.7}', metric_type)
            distance = 'minkowski';
            dpar = 0.7;
        case getIndex('mink_1', metric_type)
            distance = 'minkowski'; %default p=1, should equal to cityblock
            dpar = 1;
        case getIndex('mink_2', metric_type)
            distance = 'minkowski'; %default p=2, should equal to eucledian
            dpar = 2;
        case getIndex('mink_3', metric_type)
            distance = 'minkowski';
            dpar = 3;
        case getIndex('mink_4', metric_type)
            distance = 'minkowski';
            dpar = 4;
        case getIndex('mink_{100}', metric_type)
            distance = 'minkowski';
            dpar = 100;
        case getIndex('seucl', metric_type)
            distance = 'seuclidean';
            dpar = standard_dev_vec;
        case getIndex('chebychev', metric_type)
            distance = 'chebychev';
        case getIndex('jaccard', metric_type)
            distance = 'jaccard';
        case getIndex('mahalanobis', metric_type)
            distance = 'mahalanobis';
            dpar = covarmat;
        case getIndex('mah_1', metric_type)
            distance = 'mahalanobis';
            dpar = covarmatclass1;
        case getIndex('mah_2', metric_type)
            distance = 'mahalanobis';
```

```
                dpar = covarmatclass2;
            case getIndex('mah_3', metric_type)
                distance = 'mahalanobis';
                dpar = covarmatclass3;
            case getIndex('spearman', metric_type)
                distance = 'spearman';
            case getIndex('chisquare', metric_type)
                chisquare = @(x,Z)sqrt( sum((bsxfun(@minus,x,Z).^2) ./ bsxfun(@plus,x,Z), 2)/2 );
                distance = chisquare;
            case getIndex('jensen-sh', metric_type)
                jensenshannon = @(x,Z)sqrt( sum( x .* log((2*x)./ bsxfun(@plus,x,Z)), 2)/2 + sum( Z .* log((2*Z)./ bsxfun(@plus,x,Z)), 2)/2 );
%                jensenshannon = @(x,Z)sqrt( sum( x .* log10(2*x)./ bsxfun(@plus,x,Z), 2)/2 ) + sum( Z .* log10((2*Z)./ bsxfun(@plus,x,Z)), 2)/2 );
                distance = jensenshannon;
            case getIndex('earthmovers', metric_type)
%                %test with pdist([x; y], earthmovers) = 3.3
%                x = [0 0 0 0.2 0.3 0.5 0 0 0 0 0];
%                y = [0 0 0 0 0 0 0.1 0.2 0.7 0 0];
                earthmovers = @(x,Z)( sum( abs(bsxfun(@minus, cumsum(x./sum(x,2),2), cumsum(Z./sum(Z,2),2))), 2));
                distance = earthmovers;
%            case
%                distance = ;
            otherwise
                warning('Undefined distance metric used')
        end

end
```

### 4.5. getIndex.m

```
function i = getIndex(distance, metric_types)
    i = find(cellfun(@(x)strcmp(x, distance),metric_types));
end
```

### 4.6. Q3dataread.m

```
%% Cw2 Part 3
clc, clear

%% Read data
data = dlmread('wine.data.csv');

train_data = data(data(:,1)==1,:);
test_data = data(data(:,1)==2,:);

x_train = train_data(:,3:end);
y_train = train_data(:,2);
x_test = test_data(:,3:end);
y_test = test_data(:,2);

t_train = zeros(length(y_train), 3);
idx = sub2ind(size(t_train), 1:length(y_train), y_train');
t_train(idx) = 1;

t_test = zeros(length(y_test), 3);
idx = sub2ind(size(t_test), 1:length(y_test), y_test');
t_test(idx) = 1;

X_merged = [x_train;x_test];
train_max_idx = size(x_train,1);
test_max_idx = size(x_test,1);
Y_merged = [t_train;t_test];

%% Cross-validation
layer_mat = cell(14,1);
layer_mat{1} = 10;
layer_mat{2} = 8;
layer_mat{3} = 6;
layer_mat{4} = 4;
layer_mat{5} = [10 8];
layer_mat{6} = [10 6];
layer_mat{7} = [10 4];
layer_mat{8} = [8 6];
layer_mat{9} = [8 4];
layer_mat{10} = [6 4];
layer_mat{11} = [10 8 6];
layer_mat{12} = [10 8 4];
layer_mat{13} = [10 6 4];
layer_mat{14} = [10 8 6 4];

%% Unnormalized, tanh transfer function

best_test_acc = 0;
test_acc = zeros(5,14);
train_acc = zeros(5,14);

for i = 1:5
```

```
    for j = 1:length(layer_mat)
        [test_acc(i,j), train_acc(i,j)] = Q3NNgeneratedScript(X_merged, Y_merged,
            train_max_idx, test_max_idx, y_train, y_test, layer_mat{j}, i, 1);
        if test_acc(i,j) > best_test_acc
            best_test_acc = test_acc(i,j);
            corresponding_train_acc = train_acc(i,j);
            best_optim_setting = i;
            best_layer_setting = j;
        end
    end
end

%% Normalized, tanh transfer function

X_meanzero = bsxfun(@minus, X_merged, mean(x_train, 1));
X_norm = bsxfun(@rdivide, X_meanzero, std(x_train, 1));


best_test_acc_2 = 0;
test_acc_2 = zeros(5,14);
train_acc_2 = zeros(5,14);

for i = 1:5
    for j = 1:length(layer_mat)
        [test_acc_2(i,j), train_acc_2(i,j)] = Q3NNgeneratedScript(X_norm, Y_merged
            , train_max_idx, test_max_idx, y_train, y_test, layer_mat{j}, i, 1);
        if test_acc_2(i,j) > best_test_acc_2
            best_test_acc_2 = test_acc_2(i,j);
            corresponding_train_acc_2 = train_acc_2(i,j);
            best_optim_setting_2 = i;
            best_layer_setting_2 = j;
        end
    end
end

%% Unnormalized, sigmoid transfer function

best_test_acc_3 = 0;
test_acc_3 = zeros(5,14);
train_acc_3 = zeros(5,14);

for i = 1:5
    for j = 1:length(layer_mat)
        [test_acc_3(i,j), train_acc_3(i,j)] = Q3NNgeneratedScript(X_merged,
            Y_merged, train_max_idx, test_max_idx, y_train, y_test, layer_mat{j}, i
            , 2);
        if test_acc_3(i,j) > best_test_acc_3
            best_test_acc_3 = test_acc_3(i,j);
            corresponding_train_acc_3 = train_acc_3(i,j);
            best_optim_setting_3 = i;
            best_layer_setting_3 = j;
        end
    end
```

```
end

%% Normalized, sigmoid transfer function

best_test_acc_4 = 0;
test_acc_4 = zeros(5,14);
train_acc_4 = zeros(5,14);

for i = 1:5
    for j = 1:length(layer_mat)
        [test_acc_4(i,j), train_acc_4(i,j)] = Q3NNgeneratedScript(X_norm, Y_merged
            , train_max_idx, test_max_idx, y_train, y_test, layer_mat{j}, i, 2);
        if test_acc_4(i,j) > best_test_acc_4
            best_test_acc_4 = test_acc_4(i,j);
            corresponding_train_acc_4 = train_acc_4(i,j);
            best_optim_setting_4 = i;
            best_layer_setting_4 = j;
        end
    end
end

%% Best result separately to produce graph

[test_acc_bestsetting, train_acc_bestsetting] = Q3NNgeneratedScript(X_merged,
    Y_merged, train_max_idx, test_max_idx, y_train, y_test, layer_mat{11}, 2, 1);

disp(test_acc_bestsetting)
disp(train_acc_bestsetting)
```

### 4.7. Q3NNgeneratedScript.m

```
function [test_accuracy, train_accuracy] = Q3NNgeneratedScript(X_merged, Y_merged,
    train_max_idx, test_max_idx, y_train, y_test, hiddenLayerSize, optimFunc,
    activFunc)

setdemorandstream(391418381)

x = X_merged';
t = Y_merged';

% Choose a Training Function
% For a list of all training functions type: help nntrain
% 'trainlm' is usually fastest.
% 'trainbr' takes longer but may be better for challenging problems.
% 'trainscg' uses less memory. Suitable in low memory situations.
% help nntrain to see options

% Create a Pattern Recognition Network
%hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);

if activFunc ~= 1
    for num = 1:length(net.layers)
        net.layers{num}.transferFcn = 'logsig'; %default is tansig
    end
end

if optimFunc == 1
    net.trainFcn = 'trainscg'; % Scaled conjugate gradient backpropagation.
elseif optimFunc == 2
    net.trainFcn = 'traincgp'; % Conjugate gradient backpropagation with Polak-
        Ribiere updates.
elseif optimFunc == 3
    net.trainFcn = 'trainbfg'; % BFGS quasi-Newton backpropagation.
elseif optimFunc == 4
    net.trainFcn = 'trainrp'; %RPROP backpropagation.
else
    net.trainFcn = 'traingdx'; % Gradient descent w/momentum & adaptive lr
        backpropagation.
end

% Choose Input and Output Pre/Post-Processing Functions
% For a list of all processing functions type: help nnprocess
net.input.processFcns = {'removeconstantrows','mapminmax'};
net.output.processFcns = {'removeconstantrows','mapminmax'};

% Setup Division of Data for Training, Validation, Testing
% For a list of all data division functions type: help nndivide
%net.divideFcn = 'dividerand';  % Divide data randomly
%net.divideMode = 'sample';  % Divide up every sample
net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:train_max_idx;
```

```
net.divideParam.valInd = (train_max_idx+1):train_max_idx+test_max_idx;
% [trainInd,valInd,testInd] = ...
% divideind(size(X_merged, 1),1:train_max_idx,(train_max_idx+1):(train_max_idx+
    test_max_idx),[]);
%net.divideParam.testRatio = 0/100;

% Choose a Performance Function
% For a list of all performance functions type: help nnperformance
net.performFcn = 'crossentropy';  % Cross-Entropy

% Choose Plot Functions
% For a list of all plot functions type: help nnplot
% net.plotFcns = {'plotperform','plottrainstate','ploterrhist', ...
%     'plotconfusion', 'plotroc'};

% Train the Network
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
%e = gsubtract(t,y);
%performance = perform(net,t,y)
%tind = vec2ind(t);
%yind = vec2ind(y);
%percentErrors = sum(tind ~= yind)/numel(tind);

% Recalculate Training, Validation and Test Performance
%trainTargets = t .* tr.trainMask{1};
%valTargets = t .* tr.valMask{1};
%testTargets = t .* tr.testMask{1};
%testTargets = t_test';
%predicted_test = net(x_test');
%trainPerformance = perform(net,trainTargets,y)
%valPerformance = perform(net,valTargets,y)
%testPerformance = perform(net,testTargets,y)
%testPerformance = perform(net,testTargets,predicted_test)

% Test accuracy
pred_test = y(:, (train_max_idx+1):end);
[~,pred_class_test] = max(pred_test, [], 1);
test_accuracy = sum(pred_class_test'==y_test)/length(y_test);

% Train accuracy
pred_train = y(:, 1:train_max_idx);
[~,pred_class_train] = max(pred_train, [], 1);
train_accuracy = sum(pred_class_train'==y_train)/length(y_train);


% View the Network
%view(net)

% Plots
% Uncomment these lines to enable various plots.
```

```
%figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
%figure, plotconfusion(t,y)
%figure, plotroc(t,y)

% Deployment
% Change the (false) values to (true) to enable the following code blocks.
% See the help for each generation function for more information.
if (false)
    % Generate MATLAB function for neural network for application
    % deployment in MATLAB scripts or with MATLAB Compiler and Builder
    % tools, or simply to examine the calculations your trained neural
    % network performs.
    genFunction(net,'myNeuralNetworkFunction');
    y = myNeuralNetworkFunction(x);
end
if (false)
    % Generate a matrix-only MATLAB function for neural network code
    % generation with MATLAB Coder tools.
    genFunction(net,'myNeuralNetworkFunction','MatrixOnly','yes');
    y = myNeuralNetworkFunction(x);
end
if (false)
    % Generate a Simulink diagram for simulation or deployment with.
    % Simulink Coder tools.
    gensim(net);
end
```