Pattern Recognition Coursework 2

Theo Franquet
00967383
tcf14@ic.ac.uk

Guillaume Ramé
00978741
gr1714@ic.ac.uk

1. Question 1 - Distance Metrics

1.1. Data preprocessing

In this series of experiments, the wine dataset, consisting of 178 samples of 13 dimensions, will be used in order to better understand the differences between the various distance metrics seen in lectures. This dataset has a predefined train / test split of $\frac{2}{3}$ - $\frac{1}{3}$ amounting to 118 training points and 60 test points. We will study the recognition accuracy of nearest neighbors, implemented by *Matlab's knnsearch*, using various distance metrics, where classification error is defined as the number of misclassified test points over the total number of test points. The following experiments will be ran on both scaled (both train and test sets are divided by the standard deviation for each dimension in the train set) and the original data. It is important to note that the original data has dimensions, such as the proline content (dim. 13), which have very large standard deviation when compared to others. As we will see in the beginning of this paper, these large variations in order of magnitude will have detrimental effects on some of the studied distance metrics.

1.2. Distance metrics for experiments

Table 1 shows the various distance metrics and their formulas used as metrics for the nearest neighbor classifier used in this parts experiments. x and y represent vectors of arbitrary dimensionality, where all explicit operations $*$, $+$, $-$ and $/$ are element-wise.

Distance metric	Formula
L_1	$\sum \lvert x-y \rvert$
L_2	$\sqrt{\sum (x-y)^2}$
Chebyshev / L_∞	$\max \lvert x-y \rvert$
Cosine	$1 - \frac{x^T y}{\lvert\lvert x \rvert\rvert_2 \lvert\lvert y \rvert\rvert_2}$
Mahalanobis	$\sqrt{(x-y)^T V^{-1} (x-y)}, V = COV(x,y)$
root chi-squared	$\sqrt{\frac{1}{2} * \sum \frac{(x-y)^2}{x+y}}$
Correlation	$1 - \frac{(x-\bar{x})(y-\bar{y})}{\sqrt{(x-\bar{x})(x-\bar{x})'}\sqrt{(y-\bar{y})(y-\bar{y})'}}$
Intersection	$1 - 0.5 * (\frac{\sum min(x,y)}{\sum x} + \frac{\sum min(x,y)}{\sum y})$

Table 1. Distance metrics used in the experiments

1.3. Results

When comparing test results with the original versus normalised data, Table 2 shows that the *Minkowski* form metrics have seen the largest drops in errors from $20\%, 25\%, 28.3\%$ respectively to $8.3\%, 10\%, 11.7\%$. This is caused by the fact that they do not apply any normalization before evaluation. This causes dimensions with large orders of magnitude to over-power those with smaller magnitudes, leading to excessively large classification errors. *Histogram intersection*, which sums the minimum of the features together, similarly benefits from normalisation of the data, dropping its error rate from 20% to 11.7%.

Distance metric	Test error
$L1$	20.0%
$L2$	25.0%
Chebyshev / L_∞	28.3%
Cosine	11.7%
Mahalanobis	11.7%
Root chi-squared	8.3%
Correlation	10.0%
Intersection	20.0%

Distance metric	Test error
$L1$	8.3%
$L2$	10.0%
Chebyshev / L_∞	11.7%
Cosine	3.3%
Mahalanobis	11.7%
Root chi-squared	3.3%
Correlation	5.0%
Intersection	11.7%

Table 2. Test error as a function of distance metric used in NN classifier. Left: Without normalization. Right: With normalization

Improvements in metrics such as *root chi-squared* or *cosine distance*, while not to the same extent as previously discussed metrics, can be explained similarly, by the fact that normalization will reduce the effect of noise from the large magnitude dimensions. We can also see that *Mahalanobis* distance, which defaults in Matlab to using the covariance matrix of features, vastly outperforms the regular *Minkowski* form distances when using the original data. As expected, the normalisation done by $Mahalanobis$ means that it does not benefit from an improvement in accuracy when moving onto normalised data as it reduces to L_2 distance when the dimensions are uncorrelated. Similarly, the correlation metric, which centers and scales the feature vectors, results as expected in high accuracy rates regardless of pre-processing.

These results show how important uniform feature scaling is when using classifiers such as nearest neighbor. It is also unclear whether the curse of dimensionality is at fault for the relatively poor performance of some of the metrics in these tests, however the fact that the distribution of points in high dimensionality generally tend to a uniform distribution could contribute to the fact that traditional distance based metrics behave poorly.

2. Question 2 - K-means Clustering

The objective in this part was to employ the K-Means clustering algorithm in order to perform complexity reduction on the nearest neighbor classifier. K-Means clustering, given a desired number of centroids, iteratively clusters the dataset according to the closest centroid. The centroids are updated at every iteration according to the mean of the data points in each newly formed cluster. In essence, the K-Means algorithm strives to find centroids that minimize the within-cluster distance, where d is an arbitrary distance metric:

$$\sum_{i=0}^{n} min_{\mu_j \in C} d(x_j, \mu_i)$$

The K-Means algorithm requires, by design, the number of centroids to be learned to be specified. This can be done by scoring the quality of the clustering with itself, using metrics such as *Silhouette Score*. In this case, we have the ground truth labels, and know that there are three clusters to be learned.

Normally, the output for this algorithm is new class labels for each data point. It is however possible to use the final learned centroids to replace the training set in the nearest neighbor classification. The labels for the centroids are found by majority vote of the labels of the training points in each cluster. For each test point, nearest neighbor classification finds the closest centroid, with the label attritubuted to the test point being that of the closest centroid. In principle, this has the effect of generalising the nearest neighbor search so that the label attributed becomes that of the closest class on average. This technique also helps speed up the inference stage, as for each test point, only number-of-class distances need to be evaluated, rather than size-of-train-set evaluations (for brute force nearest neighbor). The distance metric for the NN classifier is made to match that used by K-Means (unsurprisingly, failing to match the distance metrics used by KM and NN returns accuracies of a random classifier).

The K-Means distance metrics used alongside *Matlab's kmeans* function will be L_1, L_2, $Cosine$, $Correlation$, and $Mahalanobis$. $Mahalanobis$ distance will be implemented by scaling the features with the cholesky decomposition of the inverse of the covariance matrix of the whole dataset, followed by L_2 distance. This was done for both the raw and normalized features. K-Means was repeated 5 times per distance metric, and the best arrangement, characterized by that which minimized the intra-cluster sum of distances, produced the centroids used in the next stage. Matlab's random number generator was also seeded with a constant, to make the following experiments exactly reproducible, by guaranteeing identical K-Means initialization from run to run.

2.1. Results

Distance metric	Test error
$L1$	28.3%
$L2$	28.3%
Cosine	46.7%
correlation	50.0%
mahalanobis	68.3%

Distance metric	Test error
$L1$	5.0%
$L2$	8.3%
Cosine	6.7%
correlation	6.7%
mahalanobis	6.7%

Table 3. Test error as a function of distance metric used in NN classifier, K-Means used as preprocessing step. Left: Without normalization. Right: With normalization

2.1.1 Without normalization

Left Table 3 shows the error rates as a function of the distance metric. This test first finds the three centroids that best represent the training classes according to a given metric, attributes ground truth labels to these centroids, followed by a nearest neighbor classification stage. We can see in this case that the error rates have increased by large amounts when compared to the same test without K-Means pre-processing. This can be explained by the fact that the resulting clustering, and by consequence learned centroids for each class, are biased towards the dimensions with larger orders of magnitude. K-Means will, by design like any clustering algorithm, have a tendency to separate the data into clusters along the dimensions with least variance.

The error rates of both *Cosine* and *Mahalanobis* cases are especially interesting, as they are extremely high when compared to that of the first question. With Mahalanobis, it would seem that the cluster labels returned in this case have no correlation with the ground truth labels, as the accuracy of the nearest neighbor stage is that of a random classifier. Cosine distance also seems to break down when used as the K-Means distance metric. This could possibly be due to the K-Means algorithm having trouble converging to a global minimum, instead getting regularly stuck in an unfavourable local minimum, even with multiple restarts.

2.1.2 With normalization

Right Table 3 shows the error rates for the same experiment as part 2.1.1 with the only difference being that the data is normalized before being passed through the K-Means mapping. We can see that the error rates have decreased by a large amount. This includes low error rates for $Cosine$ and $Mahalanobis$ distances, in agreement with the results seen in normalised Q1. In addition, this second set of results shows that using the class centroids as a proxy for the training set produces accuracy results that are essentially equivalent to the original method. While using class centroids rather than the training set is likely to scale much better in terms of execution speed as the size of the dataset is increased, this technique will likely become less robust than using k-nearest neighbors as the dimensionality of the data is increased, increasing the sparsity of the data in the feature space, culminating in most points (including the centroids) being approximately equidistant from other points.

3. Question 3 - Neural Network

3.1. Method

In this section, we use the same training and testing sets applied in questions 1 & 2. Input features are centered and scaled to zero mean and unit variance. This is very important when working with neural networks, as most activation functions have most of their effect very close to 0. By default, the activation function used is the rectified linear unit function $f(x) = max(0, x)$ (namely *ReLU*), however, sigmoid and hyperbolic tangent activation functions will also be applied in section 3.2.2. SciKit-Learn's *MLPClassifier* allows the easy training of fully connected networks while tweaking a wide range of parameters. The objective is to tune the ANN to achieve good classification performance on our dataset. It was chosen to use the testing set as the validation set due to the total sample size being very small. The default weight optimizing method chosen is Stochastic Gradient Descent (SGD), seen in lectures. For the following experiments, we set the learning rate to 0.01. To maximize performance while limiting degradation of training speed, we use dynamic learning rate decay, where the learning rate is divided by 5 if two consecutive epochs fail to reduce the loss function. The log-loss function is chosen for the sake of this experiment.

3.2. Initialization Settings and Learning Parameters

3.2.1 Without Hidden Layer - Linear Classifier

Figure 3 (Appendix A), which represents the first two dimensions of the LDA projection of the dataset, shows that the dataset is essentially linearly separable. It is possible to form a linear classifier with a neural network by using no hidden layers. This comes from the fact that the output layer is already a one versus rest linear classifier. This means that it should be possible to get sufficiently good test accuracy using this type of network architecture. Figure 1 shows that varying the L2 penalty has negligible impact on the test accuracy. This is due to the low number of weights present, lowering the influence of the L2 penalty. Applying our dataset to this ANN returns a test accuracy of 91%. This relatively high accuracy is yet another empirical confirmation that the dataset is linearly separable even when using the raw dimensions.

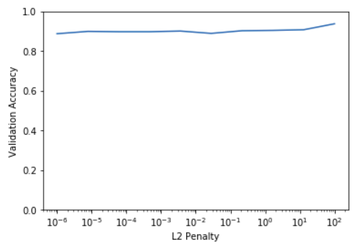

Figure 1. Accuracy vs. Varying L2 Penalty

3.2.2 With Hidden Layer(s) - Non Linear Classifier

Interesting ANN behavior can be observed according to the dimensions h of the one (or more) hidden layer(s). In our case, an ANN using **at least** 2 hidden layer neurons ($h = 2$) should be able to achieve reasonable performance, as we have shown that the data is linearly separable when projected on the first two LDA dimensions. Figures 4 and 5 (Appendix A) show the output of hidden layer neuron(s) when $h = 1$ and $h = 2$. For $h = 1$, the ANN is unable to separate the three classes of our test set. However, when $h = 2$, 2D outputs of the hidden layer clearly group data points into separable clusters. Comparing Figure 5 to the 2D LDA projection of the train set shown in Figure 3 reveals very similar characteristics. This suggests that the ANN is able to learn a linear transformation similar in idea to LDA in order to present a two dimensional linearly separable embedding to the output layer. As confirmed by Figure 6 (Appendix A), which represents the hidden layer activations post-activation-function, with overlaid learned decision boundaries, the output layer acts as a simple linear classifier.

Similarly to the previous section, it has been found that varying the L2 penalty has negligible effects on test accuracy (Figure 8, Appendix B). Figure 2 shows validation accuracies for various ANN architectures involving one or more hidden layer. We notice that a larger number of parameters results is faster convergence (fewer epochs), usually achieving better accuracy, depending on the activation function. *tanh* and *ReLU* perform similarly in all cases, achieving validation accuracies alike. When using *sigmoid* with more complex architectures (more than 1 hidden layer) however, the network has trouble converging as the validation error stays at that of a random classifier. This observation could be an illustration of the *vanishing gradient problem*[1], and is a reason why *ReLU* (or even leaky / parametric *ReLU*) is used in practise. Appendix C indicates results of the same experiment, using *adam*, a stochastic gradient-based optimizer, proposed by Kingma, Diederik, and Jimmy Ba [2], where error does converge using *sigmoid*.

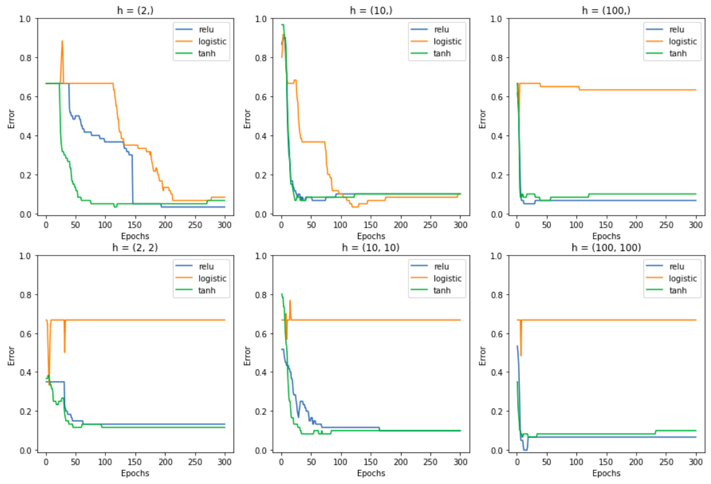

Figure 2. Testing accuracies vs. epochs, using various architectures and activation functions. *h = (neurons in layer 1, neurons in layer 2, ...)*

Looking at Figure 2, we could select a 100×100 neuron architecture, training for 15 epochs, applying *ReLU* and using $\alpha = 0.0001$ as our L2 penalty value, we achieve 100% validation accuracy. It is also possible to argument that a simple no-hidden-layer network or one with a single 2-neuron hidden layer could end up more accurate on a hypothetical test set than the more complex architectures. It is well known that a model with a large number of parameters will have a larger learning capacity and hence will more easily over-fit the training set. Table 4 in Appendix D shows that the architecture with two 100-neuron layers has over 10000 parameters, in comparison to the no-hidden-layer or single 2-neuron hidden layer networks with 42 and 37 parameters respectively. This large decrease in parameter count would potentially lead to the double benefit of a more accurate model and a model with faster inference.

3.3. Reflection and Comparison

Results have shown that equivalent or better classification accuracy has been achieved with a neural network when compared to that achieved using distance metrics in Q1 and Q2. While methods using nearest neighbor bottom out at just over 3% validation error, the larger learning capacity of the neural network enables it to achieve perfect classification. While the inference stage of the latter is more involved than the former, hence slower, accuracy of neural networks, given sufficient train data, will likely scale far better than that of the nearest neighbor methods.

[1]Andrej Karpathy: *Yes you should understand backprop* [2016, Medium]

[2]Diederik P. Kingma, Jimmy Lei Ba: *Adam: A Method for Stochastic Optimization* [2015]

Appendices

A. Neural Network Visualizations

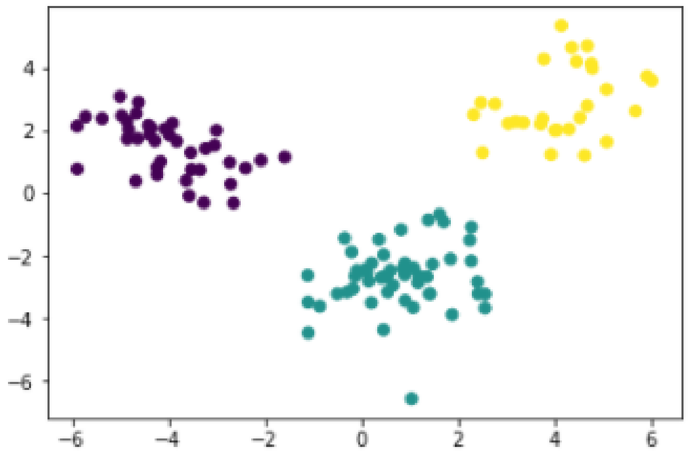

Figure 3. First two dimensions of LDA projection of wine training set

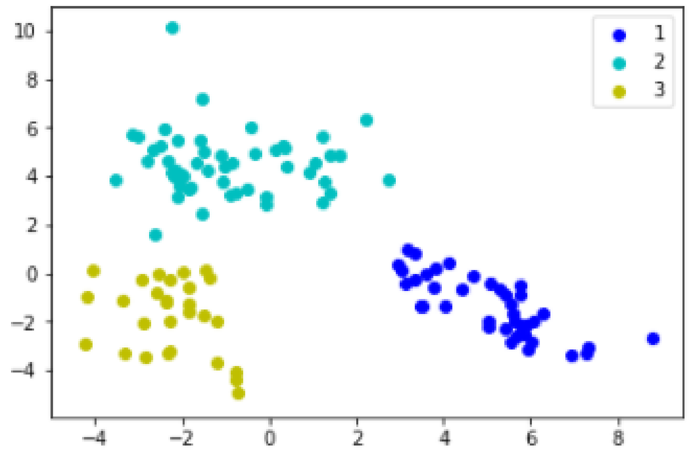

Figure 5. Hidden layer (two neurons) activations for training set before non-linearity

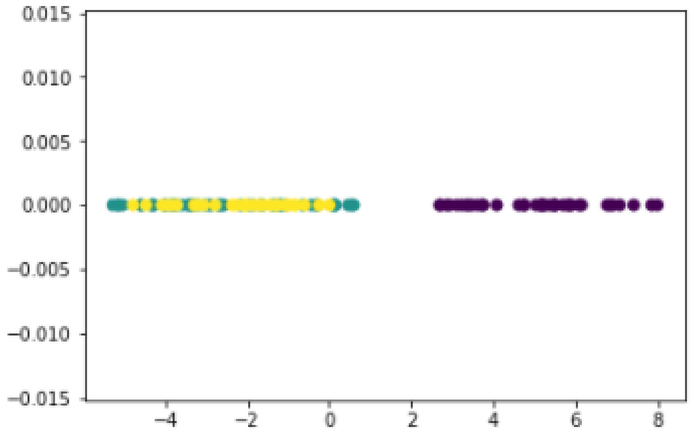

Figure 4. Hidden layer (single neuron) activations for training set before non-linearity

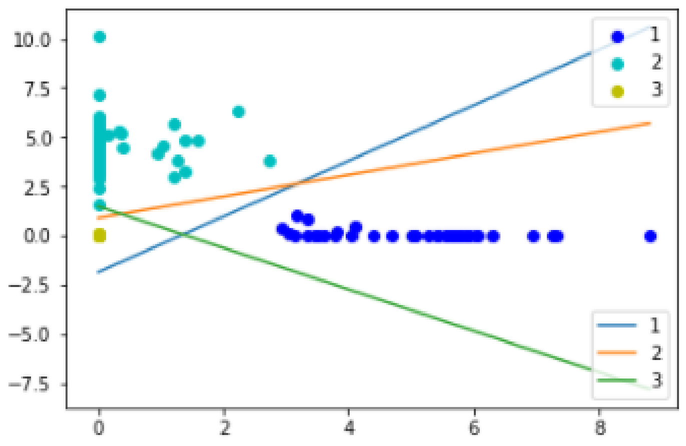

Figure 6. Hidden layer (two neurons) activations for training set after ReLU non-linearity, with decision boundaries overlaid.

B. L2 Penalty effect on multi-hidden layer NN accuracy

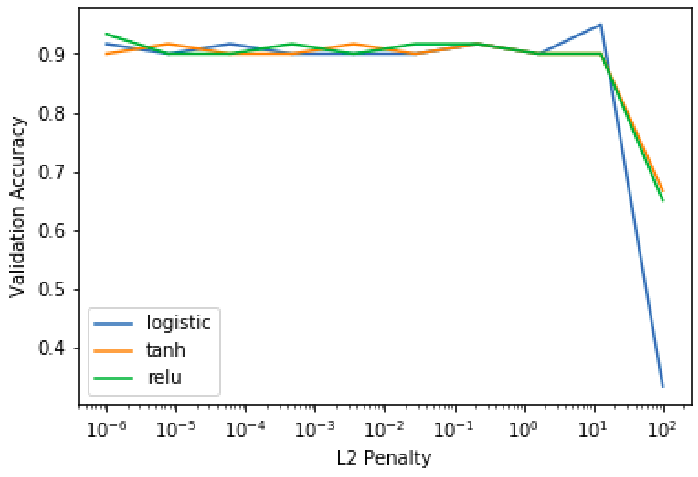

Figure 7. Accuracy vs. Varying L2 Penalty

C. Using ’adam’ solver

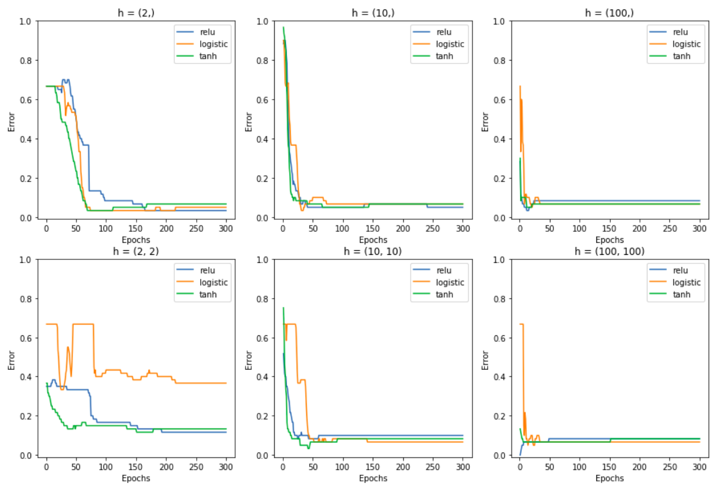

Figure 8. Testing accuracies vs. epochs, using various architectures and activation functions. *h = (neurons in layer 1, neurons in layer 2, ...)*

D. Parameter counts for the NN architecture used

Architecture	Number of parameters
$()$	$13 * 3 + 3 = 42$
$(2,)$	$(13 * 2 + 2) + (2 * 3 + 3) = 37$
$(10,)$	$(13 * 10 + 10) + (10 * 3 + 3) = 173$
$(100,)$	$(13 * 100 + 100) + (100 * 3 + 3) = 1703$
$(2, 2)$	$(13 * 2 + 2) + (2 * 2 + 2) + (2 * 3 + 3) = 43$
$(10, 10)$	$(13 * 10 + 10) + (10 * 10 + 10) + (10 * 3 + 3) = 283$
$(100, 100)$	$(13 * 100 + 100) + (100 * 100 + 100) + (100 * 3 + 3) = 10803$

Table 4. number of parameters as a function of neural network architecture