Module assignment front page

Course code and name:	FYS-3012
Candidate number:	10
Date and year:	02.11.2015

Contains 17 numbered sheets, front page included.

FACULTY OF SCIENCE AND TECHNOLOGY

Contents

1 Introduction

This report will describe the design and implementation of several pattern recognition classifiers, along with some data transformation and data reduction methods. The following algorithms have been implemented

- Least Sum of Squares
- Perceptron
- Two-Layer Perceptron
- Gaussian Naive Bayes
- Principal Component Analysis
- Laplacian Eigenmaps

We will test these methods using our own classification systems to find differences in classification results and performance.

2 Classifiers

2.1 Least Sum of Squares Classifier (LS)

When using the least sum of squares classifier we find an optimal weight vector w by finding the minimum sum of squared errors. An error e is defined as the true output y subtracted by the desired output $x^T w$ where x is a data vector in the training data X. The cost function can then be expressed as the sum of squared errors

$$J(w) = \tfrac{1}{2}\sum_{i=1}^{N}(y_i - x_i^T w)^2 \equiv \tfrac{1}{2}\sum_{i=1}^{N} e_i^2$$

where N is the number of data vectors in X. To minimize this sum we need to find when the derivative with respect to w equals zero.

$$\tfrac{\partial}{\partial w} J(w) = 0$$

$$\Rightarrow \sum_{i=1}^{N} -x_i(y_i - x_i^T w) = 0$$

$$\Rightarrow -\sum_{i=1}^{N} x_i y_i + (\sum_{i=1}^{N} x_i x_i^T) w = 0$$

$$\Rightarrow (\sum_{i=1}^{N} x_i x_i^T) w = \sum_{i=1}^{N} x_i y_i$$

$$\Rightarrow (X^T X) w = X^T y$$

$$\Rightarrow w = (X^T X)^{-1} X^T y$$

This equation to find the optimal weight vector is solved in our training function by getting the pseduoinverse of X and multiplying it with y. The pseudoinverse of X is defined as $X^+ \equiv (X^T X)^{-1} X^T$. After training the classifier with the training data, we can classify our test data by multiplying it with the weight vector and applying a sign function which gives the class label -1 if $x_i w < 0$ and 1 if $x_i w > 0$. A bias column with ones is appended to the training data and test data to shift the decision boundary away from the origin and make it more versatile.

2.2 Perceptron Classifier (P)

The Perceptron algorithm finds a minimum of the cost function with an iterative scheme called gradient descent. This scheme is used to adjust the weight vector w in the direction that decreases the cost function in every iteration step until a minimum is found.

$$w(new) = w(old) + \Delta w$$

$$\Delta w = -\mu \frac{\partial J(w)}{\partial w}$$

where μ is the learning rate that defines how large the adjustments are (usually a number between 0 and 1).

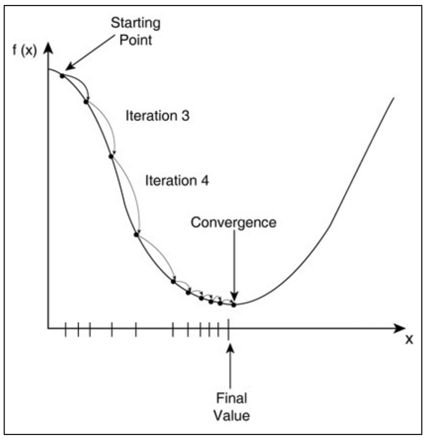


Figure 1: Gradient descent iterative scheme [1]

The perceptron cost function is defined as $J(w) = \sum_{x \in Y} \delta_x x w$
where Y is a subset of the training data with misclassified points and δ_x is -1 if $x \in w_1$ or 1 if $x \in w_2$.

2.2.1 Training

The training function initializes a weight vector $w(0)$ with random values between 1 and 0, then starts a loop that runs until a max number of iterations t is reached or the number of misclassified vectors converges to zero. A gradient vector g is initialized with zeros before iterating through the training data.

If $\delta_x(x(i)w(t)) >= 0$ $(i = 0, ..., N-1)$, we know that data point $x(i)$ is misclassified and we update the error count and gradient vector.

$g(new) = g(old) + \mu\delta_x x$

After iterating through all data points in the training data we adjust the weight vector using the gradient vector.

$w(t+1) = w(t) - \mu g$

If the number of misclassified data points is zero or max iterations is reached the loop ends, if not, the loop continues.

2.2.2 Classification

The bias and classification is done the same way as in the LS classifier.

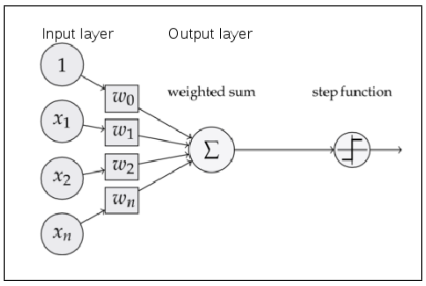


Figure 2: Abstraction of the classification operation in a basic perceptron [2]

2.3 Two-Layer Perceptron Classifier (2LP)

A Two-Layer Perceptron classifier also known as a feedforward neural network is capable of learning nonlinear decision surfaces. It can be seen as an extension to the perceptron model depicted in figure 2. The new parts is a hidden layer in the middle that can contain several units and the activation function has changed from a straight linear function to a smooth non linear function.

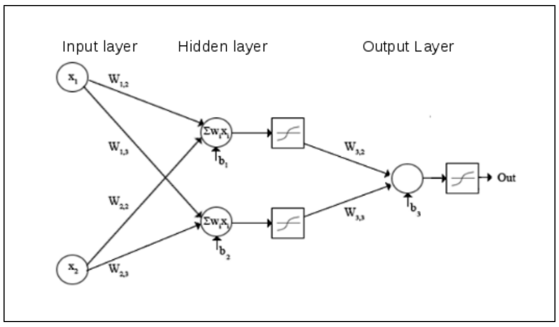


Figure 3: Abstraction of a simple two layer neural network [3]

Again we use an iterative training algorithm that computes weights to minimize a chosen cost function using gradient descent, along with another technique called backpropagation. In this case we define our cost function as the sum of errors over the whole set.

$J = \sum_{i=1}^{N} \varepsilon(i)$

$\varepsilon(i) = \frac{1}{2} \sum_{i=1}^{N} e_i^2$

$w(new) = w(old) + \Delta w$

$\Delta w = -\mu \frac{\partial J}{\partial w}$

To get a non linear decision boundary we use the hyperbolic function *tanh* as our activation function.

$f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$

2.3.1 Training (Backpropagation)

We start by initializing a random weight array and a output array for both the hidden layer and the output layer. Then we start a loop that propagates forward, propagates backward and updates the weights. The loop ends after a maximum number of epochs (iterations) or it converges to an accepted error rate.

In forward propagation each layer computes its output by multiplying its weight with the output of the previous layer and sending it through the activation function as we can see on figure 3.

$y_h = f(Xw_h)$

$y_o = f(y_h w_o)$

(h and o denotes the hidden layer and output layer respectively.)

The backward propagation starts at the last layer and computes the delta (part of the derivative in the gradient descent method) of each layer

$\delta_o = (y - y_o) f'(y_o)$

$\delta_h = \delta_o w_o^T f'(y_h)$

Then we update the weights and go to the next iteration step if nescessary.

$\Delta w_h(new) = \alpha \Delta w_h(old) + \mu x^T \delta_h$

$\Delta w_o(new) = \alpha \Delta w_o(old) + \mu y_h^T \delta_o$

$w_h(new) = w_h(old) + \Delta w_h(new)$

$w_o(new) = w_o(old) + \Delta w_o(new)$

A momentum term has been added to speed up the convergence and smooth out the oscillation of the iteration steps. This is done by simply adding the old gradient multiplied by the momentum factor α (usually a number between 0 and 1).

2.3.2 Classification

After training the classifier we can classify test data by simply sending it through the forward propagation and getting the output from the output layer. Since the output has some approximate values (very close to -1 or 1) we use the sign function round them of to the correct class labels.

2.4 Gaussian Naive-Bayes Classifier (GNB)

The Gaussian Naive Bayes classifier uses maximum a posteriori (MAP) probability estimation to find the class that is most probable for the data points in the test data. We define the probability density function $p(x|w)$ as the probability of observing x given that its class is w, in this case the features are gaussian which gives

$p(x|w_i) = \frac{1}{\sqrt{2\pi\sigma_{w_i}}} e^{(-\frac{(x-\mu_{w_i})^2}{2\sigma_{w_i}^2})} i = 1, ..., M$

where M is number of classes, μ_{w_i} is a vector with the mean of each feature in class w_i and σ_{w_i} is a vector with the variance of each feature in class w_i.

As a naive-bayes classifier we assume statistical independence between each pair of features in the dataset. This allows us to express $p(X|w_i)$ as follows

$p(X|w_i) = \prod_{j=1}^{d} p(x_j|w_i)$

where d is the number of features.

We can then use MAP probability estimation to assign an unknown sample x from the test data to a class w_m

$w_m = \underset{w_i}{\operatorname{argmax}} P(w_i) \prod_{j=1}^{d} p(x_j|w_i)$

where $P(w_i)$ is the prior probability of class w_i.

2.4.1 Training

Training the classifier is done by simply splitting the training data into classes and getting $P(w_i)$, μ_{w_i} and σ_{w_i} for each class.

2.4.2 Classification

In our classifying function we use logarithmic form for the MAP probability estimation.

$w_m = \underset{w_i}{\operatorname{argmax}} log(P(w_i) \prod_{j=1}^{d} p(x_j|w_i))$

$\Rightarrow w_m = \underset{w_i}{\operatorname{argmax}} log(P(w_i)) + log(\prod_{j=1}^{d} \frac{1}{\sqrt{2\pi\sigma_{w_i}}} e^{(-\frac{(x_j-\mu_{w_i})^2}{2\sigma_{w_i}^2})})$

$$\Rightarrow w_m = \underset{w_i}{\operatorname{argmax}}\, log(P(w_i)) + \sum_{j=1}^{d} log(\frac{1}{\sqrt{2\pi\sigma_{w_i}}} e^{(-\frac{(x_j-\mu_{w_i})^2}{2\sigma_{w_i}^2})})$$

$$\Rightarrow w_m = \underset{w_i}{\operatorname{argmax}}\, log(P(w_i)) + log(\frac{1}{\sqrt{2\pi\sigma_{w_i}}}) + \sum_{j=1}^{d} -\frac{(x_j-\mu_{w_i})^2}{2\sigma_{w_i}^2}$$

$$\Rightarrow w_m = \underset{w_i}{\operatorname{argmax}}\, log(P(w_i)) - \frac{1}{2}log(2\pi\sigma_{w_i}) - \frac{1}{2}\sum_{j=1}^{d} \frac{(x_j-\mu_{w_i})^2}{\sigma_{w_i}^2}$$

We calculate the probability estimation for all test data points for each class, compare the probability estimations, choose the largest and assign the appropriate class label.

2.5 Support Vector Machine Classifier (SVM)

The support vector machine is an algorithm that finds an optimal decision boundary by maximizing the margin between the seperation line and the data points in the training set. We see on figure 4 that support vectors are created on the furthest outliers of the class distributions. The optimal seperation line is then found by finding the maximum distance from it to the nearest data point on each side. Testing points can then be predicted to a certain class by mapping them in the same space and seeing which side they are on.

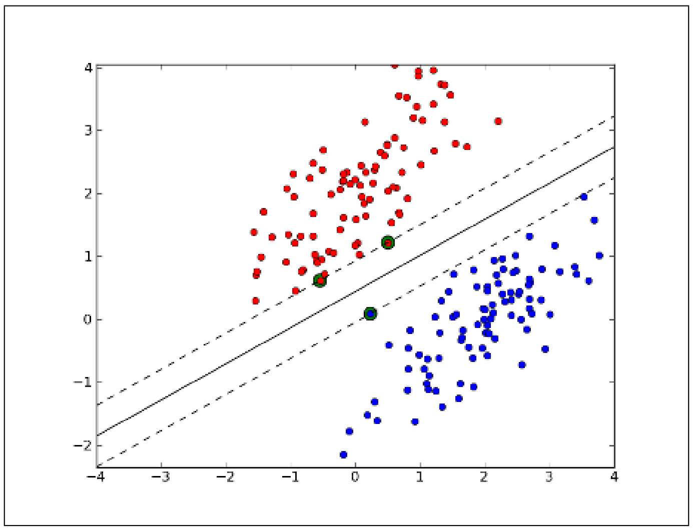


Figure 4: Support vector machine [4]

SVM has not been implemented but a linear SVM from the Scikit library [5] is included in our testing.

3 Data Transformation and Dimensionality Reduction (DTDR)

3.1 Principal Component Analysis (PCA)

Principal Component Analysis is a linear DTDR method that projects data points onto latent features which are given by eigenvectors of the dataset. [6]
The function starts by creating a mean vector containing the mean of each dimension in X

$$\mu_{1,j} = \frac{1}{N}\sum_{i=1}^{N} X_{i,j}$$

Deviations from mean is calculated by subtracting the mean vector from each row in X, this is done to center the data so we can calculate the covariance matrix.

$$B = X - \mu$$

We can then use B to find the covariance matrix of X

$$Cov = \frac{B^T B}{N-1}$$

The covariance matrix is decomposed into eigenvalues λ and eigenvectors A. These are sorted in descending order with respect to the eigenvalues, so the first eigenvector in A corresponds to the largest eigenvalue.

We choose a subset of eigenvectors $W \subset A$ corresponding to the largest eigenvalues as our basis vectors. The number of vectors in the subset is the number of dimensions we want to reduce our data too. By projecting the data onto the basis, we get the transformed matrix $T = XW$.

3.2 Laplacian Eigenmaps (LEM)

The dimensionality reduction method Laplacian Eigenmaps project points into a lower-dimensional space by using an nonlinear embedding technique based on eigenvectors. [7]
One of its main characeristics is that it optimally preserves local neighboorhood information, so that neighbors in $\mathbb{R}^d$ stay neighbors in $\mathbb{R}^l$ where $d > l$

$$\Rightarrow x \in \mathbb{R}^d \mapsto y \in \mathbb{R}^l$$

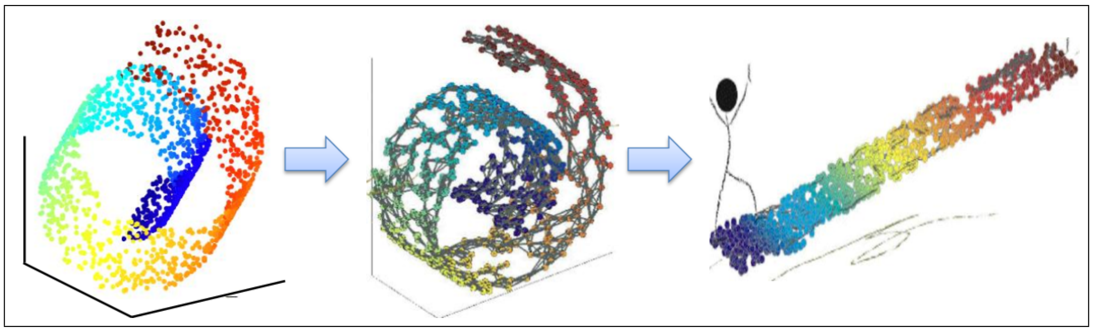


Figure 5: Laplacian Eigenmaps process

The algorithm starts by creating an adjacency matrix W for the dataset X, where each data point in X has a weighted edge between itself and its neighbours. A k-nearest neighbors (kNN) function finds the euclidean distance from a data point to all other data points in X and returns the indices and distances to its k nearest neighbors.

The weight of each edge is decided by a heat function $W_{ij} = e^{-\frac{\|x_i - x_j\|^2}{t}}(t \in \mathbb{R})$ if x_i and x_j are neighbours, if not, $W_{ij} = 0$. Using a heat function allows us to adjust the variable t according to our problem set. For example if t is a small number, the weight will be highest for very close neighbors and low for other neighbors. If t is a big number, neighbors that are not so close will also have a significant weight.

After creating the adjacency matrix, we make a diagonal matrix by summing the columns in W and use this to create the laplacian matrix.

$$D_{ii} = \sum_j W_{ji}, i = 1, 2, ..., N$$

$$L = D - W$$

We decompose L into eigenvalues λ and eigenvectors A and sort them in ascending order with respect to the eigenvalues, so the first eigenvector in A corresponds to the smallest eigenvalue. Then we can embed data points in the l-dimensional space by projecting data points onto eigenvectors associated with the smallest eigenvalues.

$$\Rightarrow x \in \mathbb{R}^d \mapsto y \in \mathbb{R}^l \text{ where } y_i = [a_1(i), .., a_l(i)]^T \text{ for } i = 1, 2, ..., N$$

We ignore the first eigenvector a_0 since it has the same embedding for all points.

4 Discussion

In the following section we test our classifiers and DTDR methods with a few different systems. This is done to see differences between our methods, look at the performance of the algorithms and perhaps discover improvements that could be made.
To stay consistent in our comparisons, all testing has been done with our default parameters.

2LP:
Units in hidden layer: 2
Learning rate $\mu = 0.1$
Momentum factor $\alpha = 0.1$
Max epochs: 5000
Accepted error rate: 0.01

P:
Learning rate $\mu = 0.1$
Max iterations: 5000
Accepted number of missclassified: 0

PCA:
Dimensions: 2

LEM:
Dimensions: 2
Number of neighbors to use $k = 4$
Heat function parameter $t = 200$

There are ofcourse circumstances where changing some of these parameters yield better results, which we will discuss.

4.1 Problem 1

A music genre classification system where we can categorize songs into genres based on its content has been implemented. The system categorizes music into two genres: rap or classical. We have a small and a big dataset which both contains the same 100 songs, but the small dataset describes songs using 4 features, while the big dataset has 101 features. A solution set with correct class labels shows that the first 31 songs in the datasets are rap and the other 69 songs are classical. The training data in each dataset consist of 20 songs. In figure 6 we can see the results when testing with the different classifiers we implemented.

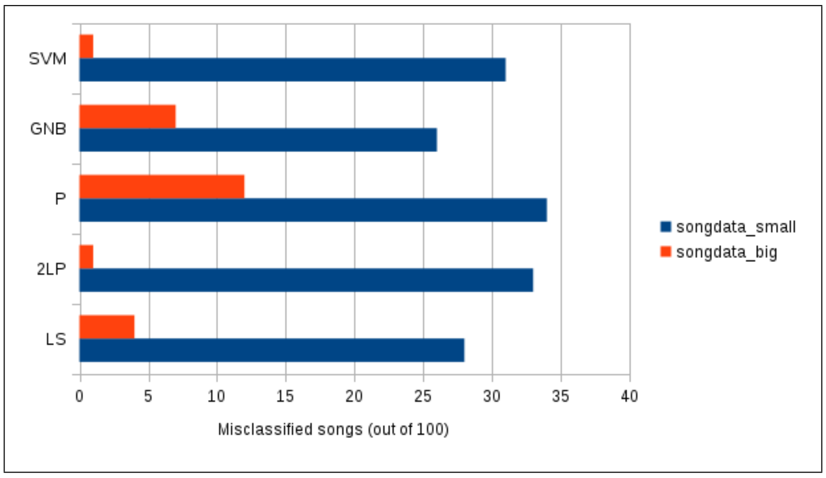


Figure 6: Test results problem 1

None of the test datasets are completely seperated with our classifiers, which means our training data does not train our classifiers sufficiently. The small dataset has around 30 misclassifications with all the classifiers, while the big dataset has 1-12. SVM and 2LP gets the best result with only 1 misclassified song, that is called 'Nocturne No.2 in E flat major Op.9 No.2' by 'Frédéric Chopin'. This song stands out the most since it gets misclassified by nearly all of the classifiers. Its a piano song that is very slow and has no beat at all, so it does not seem to have any similar traits to rap music.

The rap song that gets misclassified most often is 'Keep Watch' by 'Wu-Tang Clan'. Listening to it we can hear it has a background beat that is a little slow and very repetetive, which might be the reason it relates more towards classical music.

Since the big dataset has alot more features than the small one, it probably makes it easier to distinguish the genres. The high dimension also makes the data more sparse over a large volume space, which might make it easier to find s good decision hyperplane compared to the small dataset where the data is more dense.

4.2 Problem 2

RecoMusic is a music recommender that outputs a top 20 list of songs based on a chosen seed. A unsupervised dataset with 120 songs described by 101 features (dimensions) is used and the songs have the same classes as in the music genre classification system. The recommender creates a ranked list based on the k-nearest neighbors of the chosen seed song. e.g if we choose song number 39 as the seed, it will find the 20 songs that are closest to the seed based on euclidean distance.

We reduce the dataset to 2 dimensions using PCA or LEM, mainly because it helps against the curse of dimensionality. There are many dimensions in our data, so it becomes sparse because the volume of the space is so large, which is not preferable when using kNN algorithm to find data relations. High dimensional data also makes it troublesome to find groups of songs with properties that are alike, since there are so many different properties that can be similar or dissimilar.

The system supports visualization of results so we can see the differences in our DTDR methods and the seeds we choose. We use song number 82 as an example to find the main differences between PCA and LEM, it is a classical song called 'Capriccio Espagnol, Op.34 - 1. Alborada' by 'Nikolai Rimsky-Korsakov'.

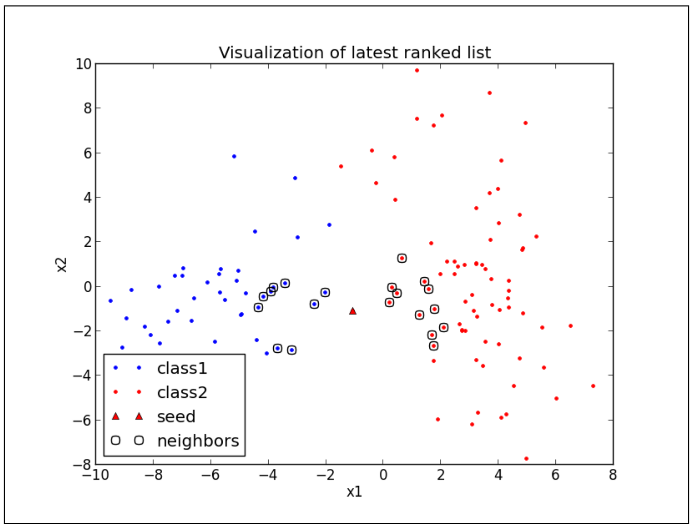


Figure 7: Song 82 ranked list visualization with PCA

Song 82 is in the middle of both class distributions which means it will choose songs from the wrong class in its ranked list. Unfortunately this applies to all outliers that are inbetween the class distributions in PCA.

The song is very upbeat with a high tempo which is probably the reason its close to the class with rap songs. Looking at the next song, it is completely opposite with a very slow tempo and a downbeat. Its on the other side of the class distribution and will only choose neighbors from its own class. We can see that PCA gives pretty good results, but it is not optimal because the data is pretty sparse and class distributions are so close.

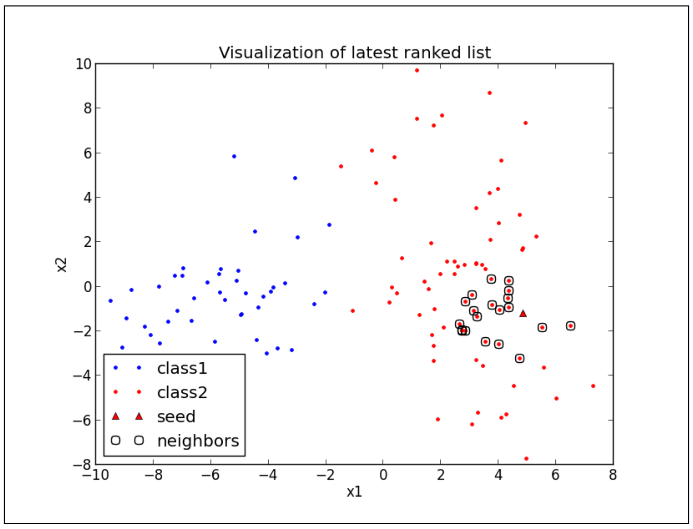


Figure 8: Song 83 ranked list visualization with PCA

On figure 9 we see the results with song 82 using LEM. It is clear that LEM optimally preserves local neighborhood information and is a far superior solution to the PCA in this case. The songs are significantly denser and the class distributions are more seperated, which makes it alot better when we want to find relevant songs for our ranked list using kNN. There are still a few outliers, but they mostly choose songs from their own class because they are closer to their own distribution then the other.

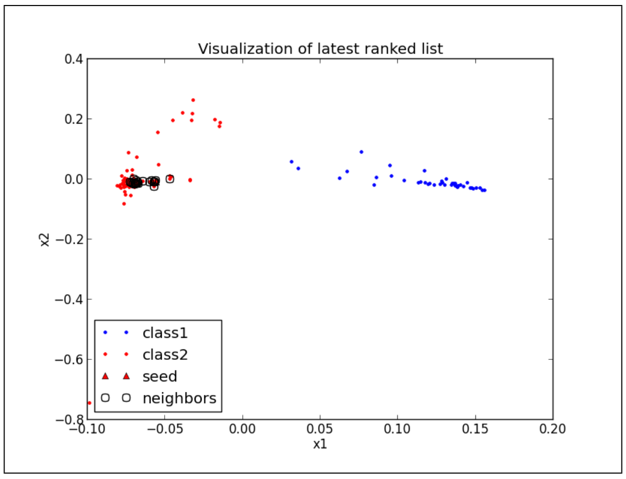


Figure 9: Song 82 ranked list visualization with LEM

4.3 Problem 3

Using a binary digit recognition system we can recognize handwritten binary numbers (0 or 1) and decode these into ASCII characters digitally. We have two training data sets available, one small with 10 data vectors (images of handwritten ones and zeros) and one large with 200 data vectors. The system will try to classify a test set with 1113 data vectors and 256 dimensions into class labels which are translated to ASCII text.

Here we have the opportunity to test all our classifier methods with and without our DTDR methods, and see how they react together. This is also a much bigger dataset compared to the ones we used previously, so algorithm performance is more noticable here. Since one of our classifiers decoded the text without any misclassifications, we used this result as a solution vector and compared the other methods to see how many errors they got. The results are shown in figure 10 and 11 (NOTE: GNB results on the original data might not be accurate, since it gave some warnings on invalid values in the data).

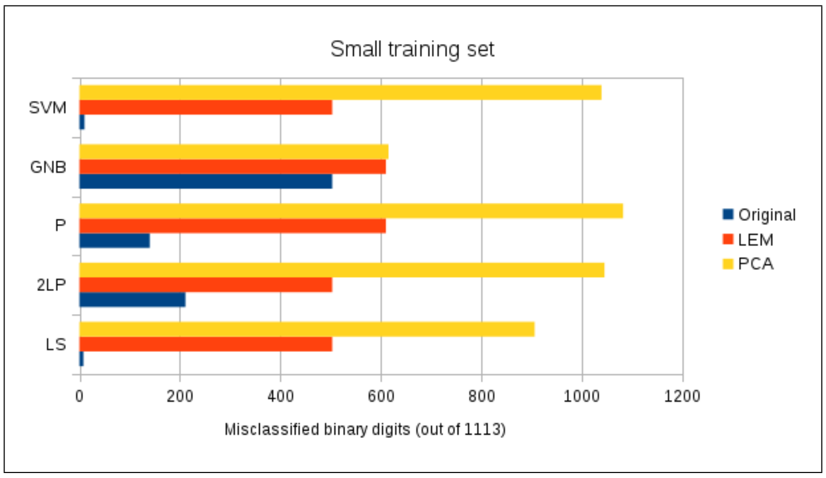


Figure 10: Decoding results with small training data in binary recognition system

A good classification was not expected with the small dataset since it only has 10 data vectors for training, but surprisingly the linear SVM and LS classifiers only got around 8 binary digits wrong when using the original data. This gave a pretty good output text that was coherent enough to understand. We can see that the other classifiers was not close with the perceptrons around 150-200 misclassified and the GNB up at 500.

When reducing the small dataset with PCA or LEM, none of the classifiers managed to get a good result. Too much important data probably went missing in the transformation. This could be fixed by increasing the number of dimensions so more data is preserved. A noticable point in the graph is that LEM gets half the misclassifications that PCA does, which most likely happens for the same reason it performed better in the recommender system. When seperating the black and white pixels in the images, LEM does a better job at densifying the data and keeping the class distributions apart by using neighboorhood information in its transformation.

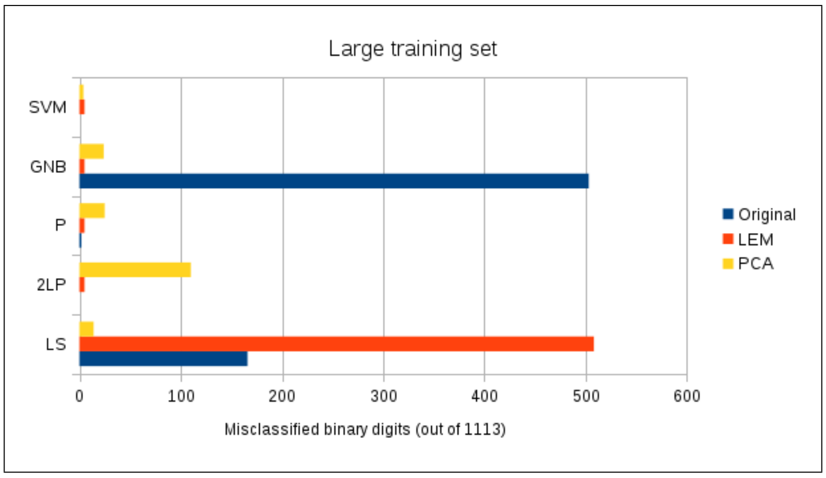
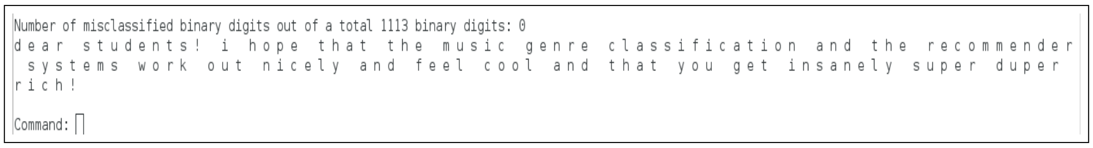


Figure 11: Decoding results with large training data in binary recognition system

The large training set has much better results then the small one, we can see that all the classifiers except GNB and LS perform well on the original data. 2LP and SVM got 0 misclassifications which gave a decoded text with no errors.

In both training sets it seems like 3 different descision hyperplanes with 5, 500 or 600 misclassified keep repeating when using LEM. The 2LP and P doesnt give a stable output since the weight vectors they produce can be different every time, so they would jump between 5 and 500 errors. Decreasing the learning rate and error rate would probably stabilize their results more. Only 5 misclassifications is still a very good result for our classifiers considering we reduced the data to 2 dimensions.

Again the PCA is a little behind LEM when it comes to getting the best possible transformed data. In this case the 2LP does not converge and gets stuck with a error rate around 20, which is why it has around 100 errors. This could easily be fixed by increasing the learning rate and momentum to get it past the slump that it is stuck in.



```
Number of misclassified binary digits out of a total 1113 binary digits: 0
d e a r   s t u d e n t s !   i   h o p e   t h a t   t h e   m u s i c   g e n r e   c l a s s i f i c a t i o n   a n d   t h e   r e c o m m e n d e r
  s y s t e m s   w o r k   o u t   n i c e l y   a n d   f e e l   c o o l   a n d   t h a t   y o u   g e t   i n s a n e l y   s u p e r   d u p e r
r i c h !

Command:
```

Figure 12: Decoded text

We can conlude that when using classifiers with DTDR methods some important data may be lost but we can still get good results and we could always increase the number of dimensions if necessary. The SVM and 2LP had the best classification results, but the 2LP could also give bad results since it does not have a stable output.

4.4 Performance and improvements

LEM clearly been better than PCA in transforming the data, but not when it comes to runtime performance. We can notice that reducing the 1113x256 dataset using LEM takes while, and it is significantly slower than the PCA. In figure 13 we can see some performance results when using LEM and PCA in the recommender system and the binary recognition system. LEM was a little bit faster on the small dataset but alot slower on the large one, so it is not optimal for big data.

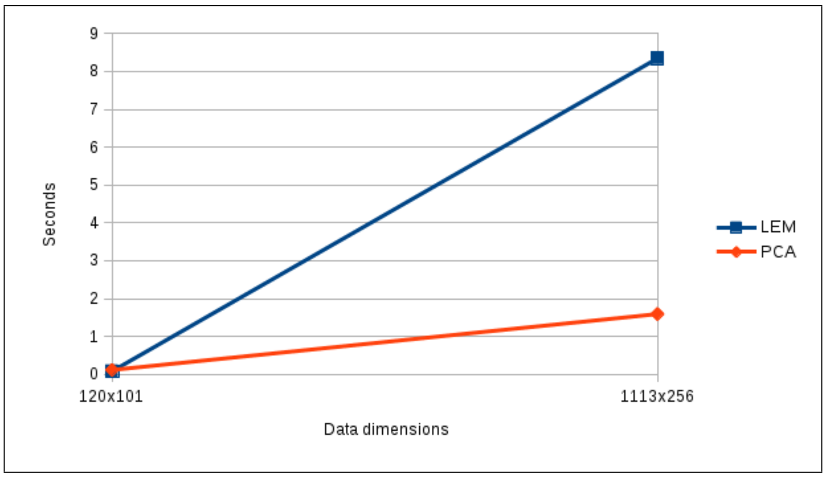


Figure 13: DTDR algorithm performance

All the classifiers performed well when it comes to runtime. The perceptrons might take a while if they dont converge, but they dont get stuck since they have a max iteration condition.

At the moment the methods we implemented are a little static to fit our systems, a big improvement would be to make them more dynamic, for example by being able to support more than two classes, different class labels etc. We could also convert our two layer perceptron to a multi-layer perceptron with some easy steps, to support adding more layers if needed. Numpy arrays has been used efficiently to avoid alot of loops in the code.

5 Conclusion

This report described the design and implementation of several pattern recognition classifiers and DTDR methods. By implementing a genre classification system, a music recommender system and a binary digit recognition system we could test out methods and find their pros and cons. Overall the SVM had the best results and 2LP was a close number two. With the DTDR methods we found out that the LEM gave better results but had worse performance than the PCA, this is a balancing issue that important to consider. There has not been any bugs with the algorithms and the systems seems to be working well.

References

[1] Gradient Descent, `http://www.yaldex.com/game-development/FILES/17fig06.gif`, 2/11/2015.

[2] Perceptron, `https://blog.dbrgn.ch/images/2013/3/26/perceptron.png`, 2/11/2015.

[3] Two-Layer Perceptron, `http://matlabgeeks.com/wp-content/uploads/2011/06/Multi-layer-perceptron.png`, 2/11/2015.

[4] Support Vector Machine, `http://www.mblondel.org/images/svm_linear.png`, 2/11/2015.

[5] Scikit's Linear SVM, `http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html`, 2/11/2015.

[6] PCA, `http://www.cs.cmu.edu/~epxing/Class/10701-11f/recitation/recitation10_qho.pdf`, 2/11/2015.

[7] LEM, `http://www.eecs.berkeley.edu/~ehsan.elhamifar/EE290A/LEM_BelkinNiyogi.pdf`, 2/11/2015.

[8] Sergios Theodoridis, Konstantinos Koutroumbas *Pattern Recognition.* Academic Press, 4. Edition, 2009.