1 Introduction

The purpose of this project was to do binary classification on hand written digits using the MNIST dataset. We implemented both a multi-layer perceptron using gradient descent and a support vector machine to classify the digits. In this report, we explain our implementation and discuss the obtained results of the two different implementations. Both implementations were done in Matlab.

2 Multi-layer Perceptron

2.1 Splitting and Preprocessing

The splitting and preprocessing step was executed only once for both datasets. The dataset has been randomized and afterwards divided into $\frac{2}{3}$ for training, resp. $\frac{1}{3}$ for validation. The input vectors of the training data set were then searched for maximum and minimum coefficient in order to normalize all datasets (training, validation and testset) according to the provided formula $\frac{x_i - \alpha_{min}}{\alpha_{max} - \alpha_{min}}$. Finally, we exported the normalized datasets to a .mat file.

2.2 The implementation of the MLP

Using an object oriented approach, we implemented a highly configurable multi layer perceptron with all paramaters being adjustable. The following parameters can be chosen:

- The number of layers of the MLP
- The number of units in each layer
- Learning rate
- Momentum term
- Number of input vectors processed at the same time
- Number of iterations

We created 3 main classes called 'Layer', 'MlpClassifier', 'MultiLayerPerceptron'.

2.2.1 Class Layer

The Layer class abstracts a layer with h_i units (adjustable). A layer is able to do the following tasks:

- Forward propagation. Given input, the layer returns the computed output with the current weights and using the transfer function $g(a_{2q-1}, a_{2q})$.
- Back propagation. Given targets for the output layer or residuals and upper layer weights, the weights are updated according to the derived formulas using gradient descent:

$$\nabla_{\omega^{(2)}} E_i = r^{(2)} * z^{(1)}$$

$$\nabla_{\omega^{(1)}_{2q}} E_i = \frac{\mathrm{d}E_i}{\mathrm{d}a^{(1)}_{2q}} * \nabla_{\omega^{(1)}} * a^{(1)}_{2q} = r^{(2)} * \omega_{2q} * a^{(1)}_{2q-1} * \sigma'(a^{(1)}_{2q}) * x$$

$$\nabla_{\omega^{(1)}_{2q-1}} E_i = r^{(2)} * \omega^{(1)}_{2q-1} * \sigma(a^{(1)}_{2q}) * x$$

2.2.2 Class MultiLayerPerceptron

MultiLayerPerceptron represents the Multi-Layer perceptron with its three phases: training, validation and testing:

- Constructor: takes as parameters the training, validation and test dataset and the training parameters such as the momentum term, the number of layers and constructs an instance of MLP

- run: does training followed by validation with a certain number of epochs and at each end of epoch computes the logistic error and the binary error of the training set and the validation set. At the end of the loop it chooses the classifier with the lowest scores and runs the test set on it. There are two different modes for using this method, one with specifying a number of epoch to perform and the other is the early stopping where the loop stops when the logistic error of the training set starts increasing.

2.2.3 Class MlpClassifier

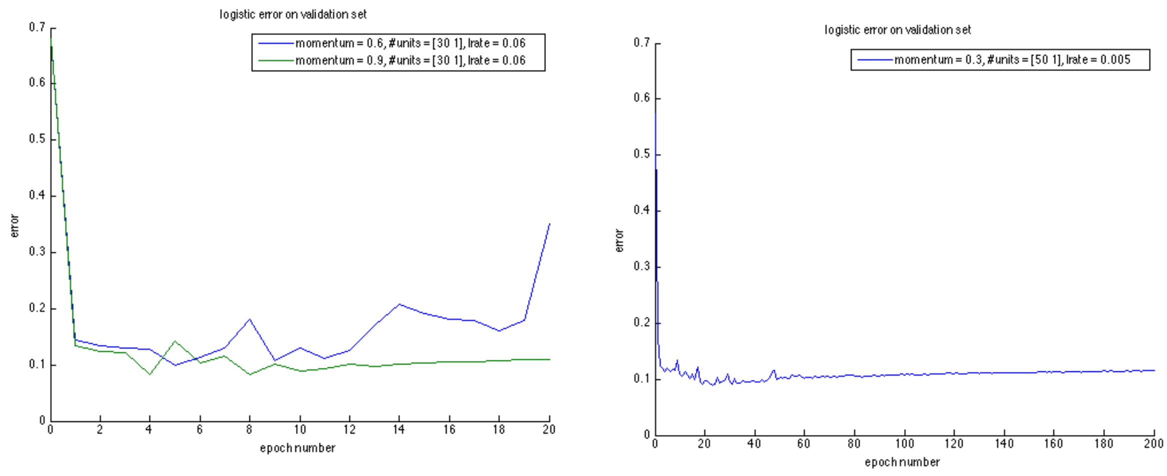
This is the final instance that can be used to classify any input vectors. The following methods can be called:

- classify: Given set of datapoints, the MlpClassifier computes the final class of each point using the underlying trained multi layer perceptron. The output is a vector containing the class for each input point.
- logErrorEval: Given a dataset with input vectors and the corresponding target vectors, the average logistic error E_{log} is computed and returned.
- binErrorEval: Given a dataset with input vectors and the corresponding target vectors, the average zero/one error E_{bin} is computed and returned. We call this error binary error.

2.3 Parameters tuning

For the choice of parameters, the number of layers is fixed at 2 and we used the stochastic gradient descent (number of input vectors processed at the same time is 1). We tried different configurations of momentum term μ, learning rate λ and number of hidden units h_1 in order to adjust the classifier.

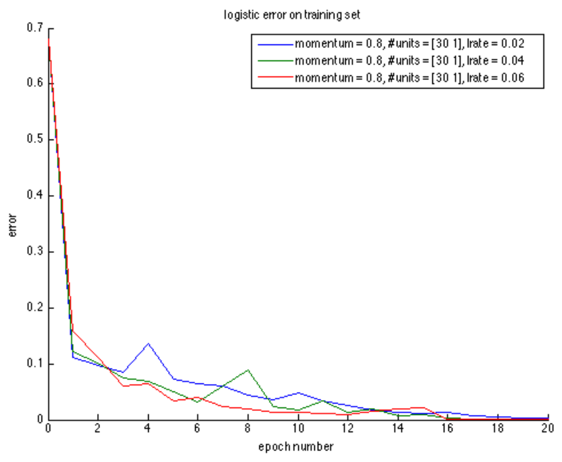
2.3.1 Momentum term μ



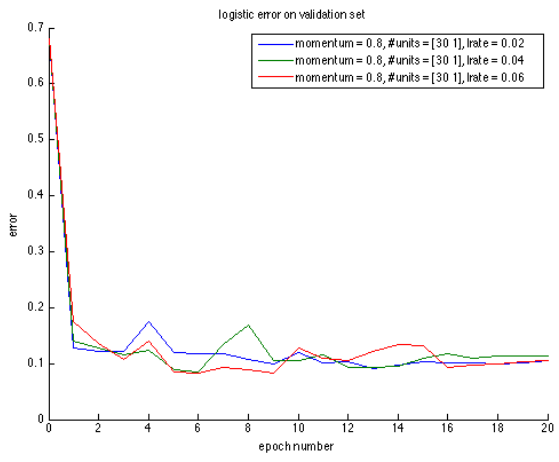
(a) Logistic error on validation set for different momentum terms (b) Example of overfitting with large number of epochs

Figure 1: Different momentum terms and an example of overfitting

Figure 1a shows that if there a low momentum term induces an effect of zigzagging in the validation set error. This is because we go too far in one direction. Using a higher momentum term smooths the curve because we "average" the directions. This prevents the algorithm to go to far in a possibly wrong direction.



(a) logistic error on training 3-5 dataset with different learning rates



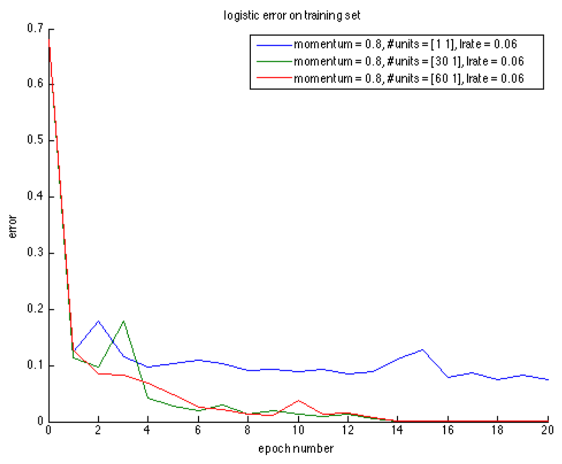
(b) logistic error on validation 3-5 dataset with different learning rates

Figure 2: Learning rates effect on logistic error

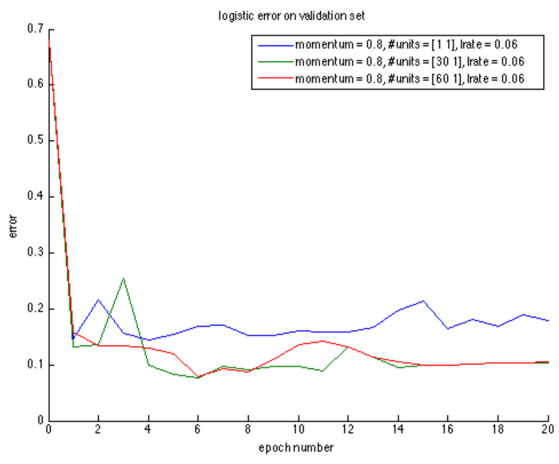
2.3.2 Learning rate λ

From Figure 2 one can notice the following: For small learning rates, such as $\lambda = 0.02$, the convergence is very slow and overfitting is not reached even after 14 epochs. As we do only small steps, we never go too far in one direction. In order to speed up the convergence, we tried a high learning rate $\lambda = 0.06$ and one can observe the contrary: fast convergence, but also much more zigzagging. We move quickly in the direction of a local minimum. Due to the fact that our step size is increased, we go too far, miss the local minimum and have to correct. This is why we observe the zigzagging. We can also notice from the graph of the training error that we start overfitting quickly (after 9 epoch). Last, we can notice that a value of $\lambda = 0.04$ is something between but rather noisy and unstable.

2.3.3 Hidden units h_1



(a) logistic error on training 3-5 dataset with different learning rates



(b) logistic error on validation 3-5 dataset with different learning rates

Figure 3: Learning rates effect on logistic error

From figure 3 we deduce the following remarks: For a small number of hidden units, $u = 1$, we notice that we cannot get a good error score in the validation and training set, simply because we cannot detect all

Momentum term:	0.8
Learning rate:	0.06
Number of hidden units	30

Table 1: Parameters for 3-5 binary classifier

Momentum term:	0.7
Learning rate:	0.06
Number of hidden units	30

Table 2: Parameters for 4-9 binary classifier

the features contained in the patterns. For a big hidden units number, $u = 60$, we notice that we converge very quickly on the training but we start doing overfitting in the validation early. This is because we stuck too much to the training data. A value of 30 hidden units behaves very similarly to the curve of the 60 units, with overfitting starting a little bit later.

2.3.4 Conclusion on parameter selection

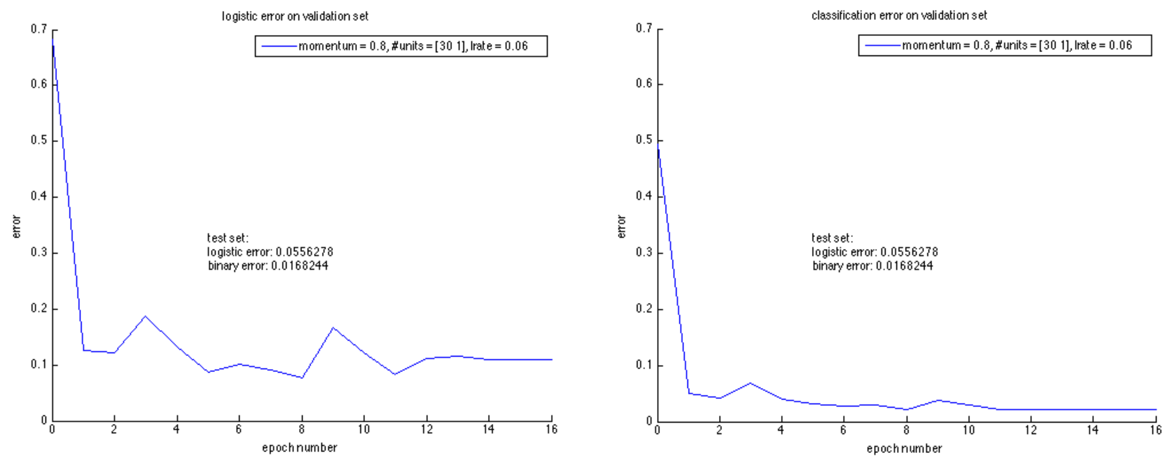
Taking into account the parameter tuning discussed in the previous part, we tuned our parameters to have speed convergence (high learning rate) without a lot of zigzagging (high momentum) and which performs well in the validation set. We choose 30 hidden units because the difference between 60 and 30 was small and 30 units computes faster. We adapted each choice for the binary classification that we have to do.

2.3.5 Overfitting example

In figure 1b we can see overfitting in the logistic error plot. We see clearly how the error increases with the number of epoches. We choose low learning rates to have smoother and better illustrations.

2.4 Results & Discussion

After choosing good parameters, we use early stopping to train our algorithm. In figures 4 and 6 one can observe the logistic error in both training and validation set for both classification problem. Figure 5 reveals the most and least misclassified patterns. Patterns in 5a and 5c are even for a human difficult to classify, whereas 5d and 5b are for humans clear. But changing only a couple of pixels, both pattern could be a the opposite as well, therefore it is comprehensible why those patterns have been misclassified.



(a) 3-5 Logistic error on validation set

(b) 3-5 Binary error on validation set

Figure 4: Performances of classifier on 3-5 datasets

We can conclude on MLP that we reach with both datasets very similar performance rates. For the different classification tasks, we used very similar hyperparameters.

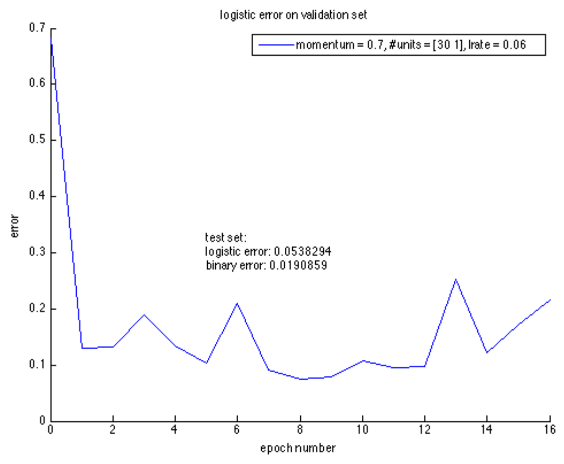
(a) 3-5 Most misclassified pattern

(b) 3-5 Least misclassified pattern

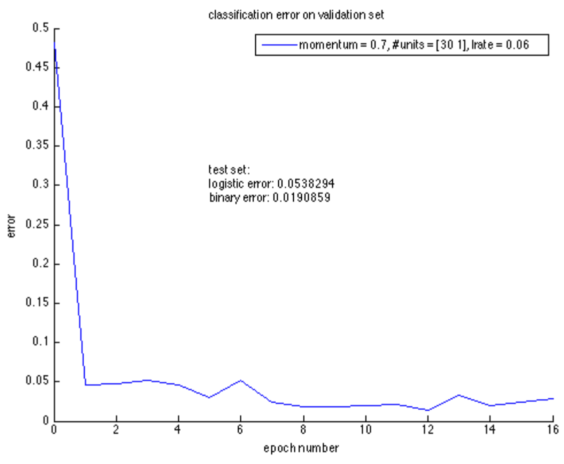
(c) 4-9 Most misclassified pattern

(d) 4-9 Least misclassified pattern

Figure 5: Misclassified patterns for 3-5 and 4-9 classification



(a) 4-9 Logistic error on validation set



(b) 4-9 Binary error on validation set

Figure 6: Performances of classifier on 4-9 datasets

3 Support Vector Machine

3.1 Methods

Our SVM implementation uses two main classes "SMOClassifier" and "SVMProject" and two additional files for preprocessing and running. Preprocessing is done the same way as for the Multi-layer perceptron but without a division of the dataset. The SMOClassifier implements the SMO algorithm we saw in the tutorial, whereas the class SVMProject implements a 10-fold cross validation and helper methods to compute the kernel and generate training and validation set for each fold.

3.1.1 Class SMOClassifier

- Constructor: Creates an SMOClassifier, initializes all the $\overrightarrow{\alpha}$, the $\overrightarrow{f}$ and computes the index sets (I_-, I_+ and I_0). Takes as an input a kernel and its corresponding targets, a C value, a τ value and a boolean for plot generation.

- doTraining: Called to train the classifier on the kernel and targets. Based on a while loop that exits once the termination criterion is reached. In every iteration, we first compute the two b_{low} and b_{up} and search for the most violated pair. We then compute the possible range $[L, H]$ and $\eta = K_{ii} + K_{jj} - 2 * K_{ij}$. Depending on the value of η, we compute the new α_j differently (if η is too small, we have a division by "almost" zero). Once the new α_j is computed, we can deduce the new α_i and update the corresponding index sets and $\overrightarrow{f}$. The bias factor b is deduced from b_{low} and b_{up} during the last iteration.

- getTrainingError: Public method to compute the training error. Can only be called once the classifier has been trained.

- classify: Once training is done, we can classify new unknown input vectors. We first need to compute a new Kernel from the new input matrix X and the training matrix M. We first compute the norm of each input vector $\mathbf{d_1} = [||x_i||] \in \mathbb{R}^k$ and the norm of each training vector $\mathbf{d_2} = [||m_i||] \in \mathbb{R}^n$. We then can simply compute the matrix A and the kernel K:

$$A = \frac{1}{2} * \mathbf{1}_k \mathbf{d_1}^T + \frac{1}{2} * \mathbf{d_2} \mathbf{1}_n^T - X * M^T$$

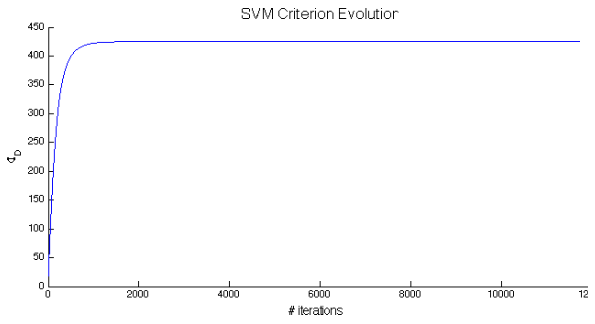
$$K = \exp(-\tau * A)$$

Finally, the output is derived using the newly computed kernel, the trained $\overrightarrow{\alpha}$, the target vectors $\overrightarrow{t}$ from the training set and the bias b.

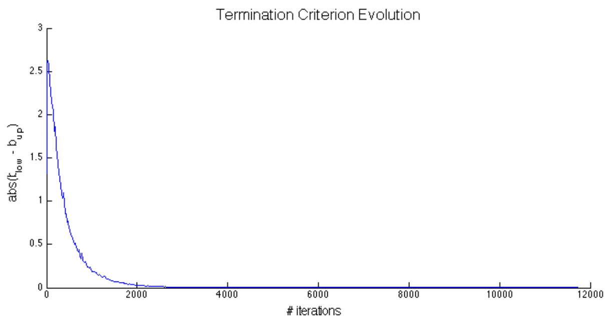
3.1.2 Class SVMProject

Only the implemented cross validation is explained in detailed form here, as the rest of the class are simple helper methods. We have an outer loop over all possible values of τ and a nested loop over all possible C. For each of those combinations, we do a 10-fold cross-validation: First, the dataset is split in 10 parts. Each part is used 9 times for training and once for validation. After training, we use the validation set to compute the error (one / zero). All 10 errors are summed up and averaged. After all combinations have been tested, the configuration which produced the minimal error is chosen to train a classifier using the full training dataset. Finally, we test the classifier using the test dataset and report its performance.

3.2 Results & Discussion



(a) SVM Criterion with number of iterations



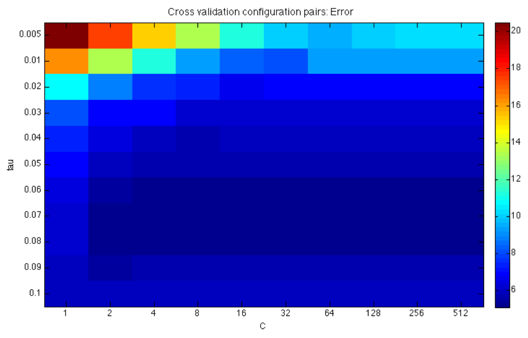
(b) Termination criterion with number of iterations

Figure 7: Learning rates effect on logistic error

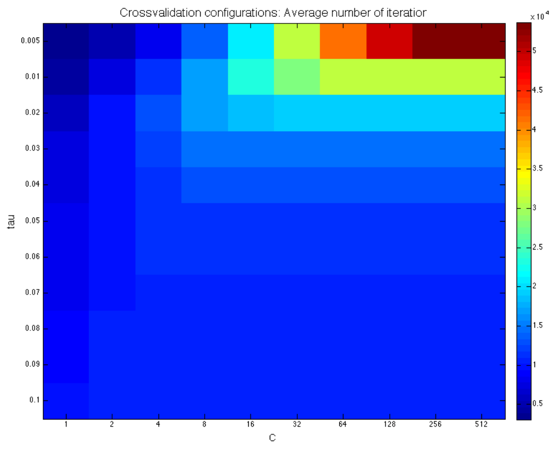
The first plot in Figure ?? shows the SVM criterion Φ_D the SMO algorithm is maximizing.

$$max_\alpha \left\{ \Phi_D(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j t_i t_j \phi_i^T \phi_j = \mathbf{1}^T \alpha - \frac{1}{2} \alpha^T (diag\mathbf{t}) \mathbf{K} (diag\mathbf{t}) \alpha \right\}$$

Already after about 2000 iterations, the criterion does only very small changes (order of 1e-4). In the second plot of Figure ??, we can observe that the difference between b_{low} und b_{up} falls to the order of 1e-2 at the same number of iterations. From there on, only small adjustments on the α's are needed for the fulfillment of the Karush-Kuhn-Tucker conditions.
In Figure 8a one can see the average number of errors produced in each fold of a 10-fold cross-validation for different sets of C and τ values. Dark blue areas show configurations that produces less average error, while other colored areas stand for higher average error. We clearly see that one can expect higher error numbers with low τ values. One can also observe that C values do not have high influence on the number of produced errors. For high τ values, such as 0.6 or 0.7, all C starting at 2 produce the same average error. One can see that this "threshold" goes up with lower τ values. Our cross-validation system choose the $C = 2$ and $\tau = 0.07$ as the optimal configuration. After learning with the complete training set,



(a) 10-fold cross-validation. Plot of the average number of errors using different (C, τ) configurations.



(b) 10-fold cross-validation. Plot of the average number of iterations using different (C, τ) configurations.

Figure 8: Example of overfitting on validation dataset

the performance on the test set was very high: 17 out of 1991 were wrong classified. This leads to a performance of 99.15%. We observed that the number of iterations changed depending on the C and τ values. We plotted the average number of iterations during cross validation for each of the tested (C, τ) configurations. One can clearly observe that with low τ and low C, only a very small number of iterations is needed until the KKT-conditions are fulfilled. On the other hand, low τ and high C lead to extremely high numbers of iterations. Finally, one can deduce from Figure 8 that the configuration ($C = 2$, $\tau = 0.07$) is reasonable in both number of iterations (and therefore time) and number of errors.

3.3 Comparison MLP and SVM

After running the tests, we conclude that SVM was both more efficient, faster and easier to handle. With SVM, we reach a performance of 99.15%, while MLP reaches only about 98.1%. We see the performance difference also in the training set. In general, it was easier to find optimal values for SVM than for the MLP: With MLP, using same values, the performance had high variance. If we rerun the training multiple times, because of the permutation of the input vectors, the result can be slightly different.

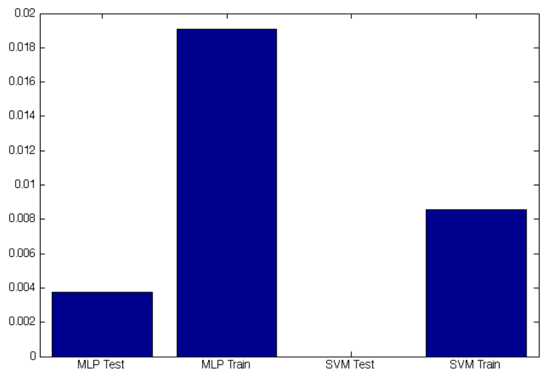


Figure 9: Relative Comparison between SVM (training, test) and MLP (training, test). Training set has input size 6000, Test set has 1991