BMI coursework

Team Error404: Georgios Gryparis, Agnese Grison, Charalambos Hadjipanayi, Edoardo Occhipinti

Dept. of Bioengineering, Imperial College London

Abstract — **In this report the neural spikes of a monkey reaching 8 different target directions were decoded to predict the x and y hand positions of the movements. Three different classifiers were implemented: k-Nearest Neighbors (k-NN), Support Vector Machine (SVM) and Bayesian. Majority voting was used to select the most likely direction, obtaining a final angle classification accuracy of 99.75%. Principal Component Regression (PCR) was used to estimate the x and y hand positions as a function of time obtaining a Root Mean Square Error (RMSE) of 7.37 ± 0.67 with the optimal number of principal components.**

I. Data & Cross validation

The data (**D**) used to develop our algorithm consists of *800* experiments. Each experiment *n* contains a trajectory of 2D hand motion (**H** = [$\mathbf{H_X}$, $\mathbf{H_Y}$]), along with the neural activity from *98* neural units (**S**) sampled at 1ms intervals. Experiments can be distinguished into *8* distinct classes (**C**) representing the reaching angle of the trajectory. In **D**, there are 100 experiments for each class $C_k \in \boldsymbol{C}$. In order to draw meaningful relations between **S** and **H**, the neural data must be reduced to a feature vector. Two feature representations were investigated, both based on firing rates. The first representation ($\tilde{\boldsymbol{f}}(\tau)$) was constructed by computing the average firing rate of each of the 98 neural units from the beginning of the experiment (time t=0) until the time of interest (time t = τ). The second, higher dimensional representation ($\boldsymbol{f}(\tau)$) was constructed by extracting the 98 spike trains from time t = 0 to t = τ, then splitting each into bins of size δt, computing the within-bin firing rates and concatenating the results into a single vector. The feature extraction process is illustrated in Figure 1.

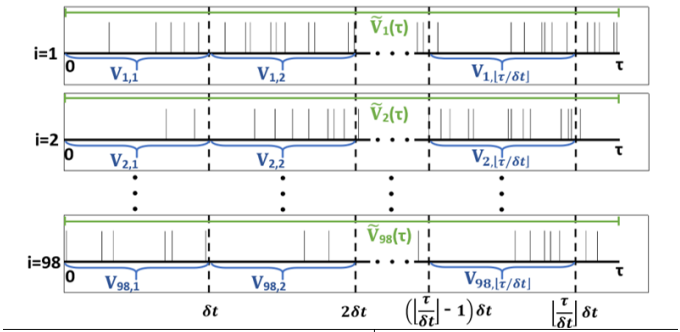


Figure 1 - Feature extraction process for both representations. Floor operator is denoted by ⌊·⌋. Analytical expressions are described in Appendix A.1.

Preliminary results showed that representation $\boldsymbol{f}$ outperforms $\tilde{\boldsymbol{f}}$ in both classification accuracy (Appendix B) and RMSE of the predicted trajectories (Appendix A.9). Thus, representation $\boldsymbol{f}$ is used for the rest of this report, unless otherwise specified.

As the position estimation was performed at times that are multiples of 20ms, bin size δt=20ms was used for trajectory regression. For classification, bin sizes of δt=20ms, 40ms and 80ms were used. Note that updating the classification can only be performed at times that are multiples of δt and that feature vector size decreases with larger δt and increases with larger τ.

The dataset (**D**) was split, creating a training set ($\mathbf{D_{train}}$) and a testing set ($\mathbf{D_{test}}$). These were generated randomly for a given ratio $s_p = N_{train}/(N_{test}+N_{train})$, while ensuring that the size of each class within each set ($N_{C,train}$ and $N_{C,test}$) was constant, to allow for cross-validation (performed using the Monte-Carlo method, see Appendix A.13). Final evaluation of the algorithm was performed using an 82-trial dataset provided by Dr Clopath.

II. Tuning Curves & Population Decoding

Neurons broadly encode movement direction and orientation. Tuning curves that plot firing rate vs direction angle for the training data can be used to determine each neuron's preferential direction. The direction of movement can then be theoretically inferred by the preferred directions of neurons, weighted by their respective firing rates [1]:

$$\begin{pmatrix}\cos(\theta)\\ \sin(\theta)\end{pmatrix} = \sum_i \frac{r - r_0}{r_{MAX}} \begin{pmatrix}\cos(p_i)\\ \sin(p_i)\end{pmatrix}$$

where θ is the predicted direction and p_i is the preferred direction for the i^{th} neuron (direction at which its tuning curve peaks). r is the current firing rate, r_0 is the baseline firing rate (tuning curve minimum) and r_{MAX} is the peak firing rate.

To optimize population decoding, the neurons that best represent each preferred direction had to be determined. This was achieved by inspecting the normalized and unnormalized tuning curves and quantifying the noise and the strength of the preferred direction. Population decoding was performed at t=320ms to 560ms, which represents the full range of possible classification times since the shortest trajectory in the dataset is 571ms. For s_p=0.7, over 20 iterations, mean angle classification accuracy peaked at t=480ms and was 49.25% (Appendix A.2).

III. Average Trajectory

A naïve method of approximating the hand position $\mathbf{H_{test}}(t)$ is to estimate that it is equal to the average hand position at time t of all experiments that belong to the same class. The mean trajectory of class C_k, $\bar{\boldsymbol{H}}^{Ck}(t)$, can be estimated by:

$$\bar{\boldsymbol{H}}^{Ck}(t) \approx \bar{\boldsymbol{H}}^{C_k}_{\boldsymbol{train}}(t) = (1/N_{C,train}) \sum_{n\in Ck} \boldsymbol{H}^n_{\boldsymbol{train}}(t)$$

where n denotes the experiment index. In order to compute the average, it was necessary that all trajectories covered the same length of time. Therefore, the longest trajectory was found for each angle and the shortest trajectories were padded with the last x and y hand position values. This method, instead of zero padding the shorter trajectories, ensured that the average trajectory did not suffer from erroneous average values or a reduction in data length. This estimation technique was used to benchmark the different classifiers. Note that given perfect classification, this method gives an RMSE of 6.96 with standard deviation 0.56 over 100 iterations for s_p = 0.7.

IV. PCA-LDA & Data Visualization

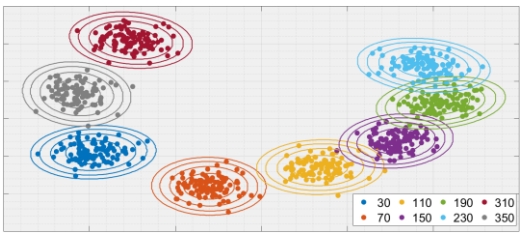


Figure 2 – Features with δt=80ms, at t=320ms, using PCA-LDA (M_{PCA}=170, M_{LDA}=2). Contours of fitted Gaussian distributions included for illustration.

Using Linear Discriminant Analysis (LDA) and Principal Component Analysis (PCA), separation between classes can be maximised. LDA is a tool used to transform data, by rotating the feature space, to find the optimal direction with respect to class separation. To improve LDA performance, PCA was first used to reduce the dimensionality of the feature space. To visualize the data, the feature vectors were projected onto a 2D subspace (Figure 2). On each cluster a bivariate Gaussian distribution was fitted using its mean and covariance matrix.

V. K-NN CLASSIFIER

To predict the direction of movement, a simple k-NN algorithm was employed. At time t, for each feature vector $\boldsymbol{f_{test}}(t)$ in $\mathbf{D_{test}}$, the k nearest neighbours were determined from the feature vectors $\boldsymbol{f_{train}}(t)$ in $\mathbf{D_{train}}$ using Euclidean distance. Then, the predicted direction was the majority vote of the labels of the neighbours. Alternatively, the centroid of each class within the training set $\bar{\boldsymbol{f}}^{Ck}_{\boldsymbol{train}}(t) = (1/N_{C,train}) \sum_{n \in Ck} \boldsymbol{f}^{n}_{\boldsymbol{train}}(t)$ can be calculated and each testing feature vector assigned with the label of its nearest centroid. The second method produced improved results, both in terms of accuracy, as it removes noise, and classification time (Figure 3) since it requires a smaller number of comparisons to make a prediction.

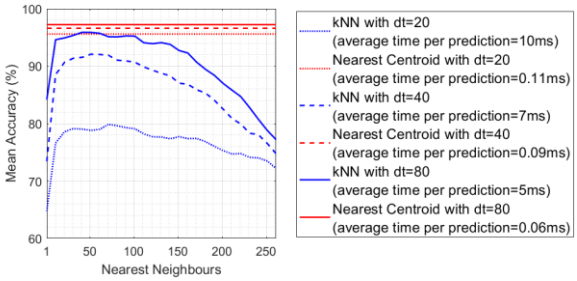


Figure 3- Accuracy vs # Nearest Neighbors (t=320ms, s_P=0.7, 20 iterations). Location of peak is related to N_{train} (for s_P=0.7, N_{train}=560)

The nearest centroid classifier was optimised by finding a transformation that maximally separates classes in $\mathbf{D_{train}}$:

$$\boldsymbol{\phi_{train}}(t) = \boldsymbol{W}(\boldsymbol{f_{train}}(t) - (1/N_{train}) \sum_{n \in \boldsymbol{D_{train}}} \boldsymbol{f}^{n}_{\boldsymbol{train}}(t))$$

Then, the centroids $\bar{\boldsymbol{\phi}}^{C_k}_{\boldsymbol{train}}(t) = (1/N_{C,train}) \sum_{n \in C} \boldsymbol{\phi}^{n}_{\boldsymbol{train_{\delta t}}}(t)$ can be found and the testing feature vectors transformed $\boldsymbol{f_{test}}(t) \rightarrow \boldsymbol{\phi_{test}}(t)$ so that each testing feature vector can be assigned with the label of its nearest centroid in the transformed feature space. To find **W**, PCA-LDA was used with a fixed number of LDA bases (M_{LDA}=7) and a varying number of PCA bases (optimal M_{PCA} was deduced empirically). Details on PCA-LDA can be found in Appendix A.3.

The M_{PCA} hyperparameter was optimized with respect to minimizing the RMSE produced by using the average trajectory. This metric was chosen since raw accuracy numbers were insufficient to determine the optimal classification method (trajectory estimation error is also dependent on the time of correct classification). A large range of M_{PCA} produced mean RMSE results within 0.1 of the local minimum for s_P=0.7, using 20 iterations (Appendix A.4). The results for M_{PCA} that minimize RMSE using different δt are tabulated below.

Mean Accuracy (%)

Time of last Classification (ms)	A	B	C	D	E	F
320	95.6	96.1	96.6	97.3	97.1	98.2
340	96.2	96.9	96.6	97.3	97.1	98.2
360	96.9	97.6	97.5	97.7	97.1	98.2
380	97.5	97.6	97.5	97.7	97.1	98.2
400	97.5	97.8	98	98.3	98.6	99.3
420	97.6	97.7	98	98.3	98.6	99.3
440	97.6	98.2	98.1	98.8	98.6	99.3
460	97.8	98.4	98.1	98.8	98.6	99.3
480	97.6	98.4	98.2	99	98.7	99.4
500	97.7	98.5	98.2	99	98.7	99.4
520	97.4	98.2	98.3	99.2	98.7	99.4
540	97.6	98.3	98.3	99.2	98.7	99.4
560	97.8	98.6	98.4	99.3	99	99.5

mean RMSE (Average Trajectory)

A	B	C	D	E	F
13.2	12.46	11.95	11.01	11.24	9.76
12.5	11.61	11.95	11.01	11.24	9.76
11.74	10.79	10.92	10.71	11.24	9.76
11	10.7	10.92	10.71	11.24	9.76
10.9	10.39	10.26	9.88	9.3	8.04
10.77	10.48	10.26	9.88	9.3	8.04
10.77	9.96	10.12	9.03	9.3	8.04
10.69	9.67	10.12	9.03	9.3	8.04
10.85	9.63	9.97	8.83	9.31	7.98
10.74	9.57	9.97	8.83	9.31	7.98
11.01	9.84	10.01	8.65	9.31	7.98
10.88	9.77	10.01	8.65	9.31	7.98
10.79	9.63	9.88	8.48	9.11	7.96

Figure 4 –Accuracy (%) and RMSE (Average Trajectory) using Nearest Centroid. A: δt=20, no PCA-LDA. B: δt=20, MPCA = 35. C: δt=40, no PCA-LDA. D: δt=40, MPCA=260. E: δt=80, no PCA-LDA. F: δt=80, MPCA = 170. (MLDA=7 in all cases).

The best result (RMSE using average trajectory: mean = 7.96, std = 0.79; over 50 iterations) occurs when M_{PCA}=170 and the label is updated at t=320, 400, 480 and 560ms (δt=80ms). Further improvements on this classification method would involve using other distance metrics (Manhattan Distance, cosine distance, Earth mover distance), and varying M_{LDA}.

VI. SUPPORT VECTOR MACHINE CLASSIFIER

The SVM method aims at finding a hyperplane in the feature space which allows for maximum linear separation of two classes of data, thus creating an optimal linear decision boundary for binary classification.

Multiclass classification was performed using a decision tree composed of three layers, as illustrated in Figure 5. In the first layer (Classifier 1), the input angle was classified as "right" (310° to 70°) versus "left" (110° to 230°). Separating the angles in Classifier 1 between "left" and "right" was informed by Figure 2. In the second layer, the angle was classified into further sub classes (groups of two angles) using Classifiers 2a and 2b. Finally, the input was assigned a specific predicted angle using the nearest centroid method described in Section V, only differing in that the only two centroids used were the ones corresponding to the two possible binary classification options.

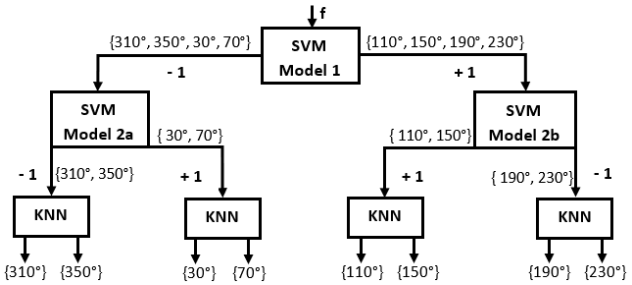


Figure 5 - SVM decision tree

SVM was implemented using a baseline code provided by Dr. Clopath. To minimize the risk of overfitting due to outliers while ensuring margin was as wide as possible, a soft margin rule was implemented to relax the SVM optimization. A factor *c* was incorporated to represent the degree of misclassification allowed. This had the added benefit of increasing the method's generalization capability. Initially, the Linear Kernel method was used. Despite the fast computation of the Linear classifier, classification performance was not satisfactory (less than 90% mean classification accuracy at t=320ms, for s_P=0.7, details in Appendix A.5). For this reason, a non-linear kernel was used, specifically the Gaussian kernel, which allows for non-linear mapping of data into higher dimensional space, where it is more likely to be linearly separable. This non-linear approach is theoretically expected to be at least as good as the linear kernel [2]. A parameter representing the spread of the Gaussian (σ) was used to control over- and under-fitting of the training set (lower σ results in tighter decision boundary).

Preliminary results (Appendix A.5) showed that, as in the case of k-NN, using only the mean feature vector of each class to train the nonlinear SVM resulted in significant improvement in classification accuracy (as compared to training on all feature vectors). Thus, the SVM training input was reduced to an 8-column matrix, which also reduced computation time. When the mean feature vectors were used, in the final layer there were only two training data points. Hence, k-NN and SVM decision boundaries were equivalent, but k-NN was faster. This method based on the Gaussian kernel produced superior results than the linear kernel as seen in Appendix A.5.

This method was used both with feature vectors **f** and the PCA-LDA transformed feature vectors $\boldsymbol{\phi}$. To determine the parameter σ: *c* was set to 1, the unmodified feature vectors (**f**) were used, σ was varied and the mean accuracy and RMSE using mean trajectory of the SVM method were recorded over 20 iterations. Then, under the same conditions, σ was set to the optimal value and *c* was varied. Additionally, the M_{PCA} hyperparameter was optimised for RMSE using mean trajectory with the transformed feature vectors $\boldsymbol{\phi}$ (Appendix A.6). Results for optimal *c*, σ and M_{PCA} (50 iterations) are shown in Figure 6.

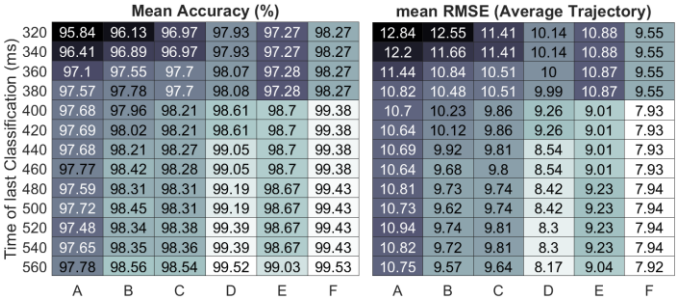


Figure 6 - Accuracy and Mean (Average Trajectory) using SVM with Gaussian Kernel. A: δt=20, σ=0.1, no PCA-LDA. B: δt=20, σ=0.1, MPCA=85. C: δt=40, σ=0.1, no PCA-LDA. D: δt=40, σ=0.1, MPCA=190. E: δt=80, σ=0.07, no PCA-LDA. F: δt=80, σ=0.07, MPCA=180. (MLDA=7, c=1 and s_P=0.7 in all cases). c (0.1-10 range) does not affect results and σ (0.05- 0.2 range) only slightly affects results (Appendix A.6).

The best result (RMSE using average trajectory: mean = 7.92, std = 0.80; over 50 iterations) occurred when σ=0.07, c=1, M_{PCA}=180 and the label was updated at t=320, 400, 480 and 560ms (δt=80ms). Further improvements on this method would involve tuning the c and σ parameters together rather than separately, as well as optimising them independently in each decision tree layer and for different values of M_{PCA} and M_{LDA}.

VII. Bayesian Classifier

This method computes the conditional probability $p(C_k|\boldsymbol{f})$ of the feature vector $\boldsymbol{f}$ belonging to each class C_k and assigns $\boldsymbol{f}$ to the class with highest probability. To train the classifier, a model was created for each angle at each time from 320ms to 560ms in δt time steps. Each model returns the probability of feature vector $\boldsymbol{f}$ given a class C_k, i.e. $p(\boldsymbol{f}|C_k)$. The Bayesian classifier assumes that the data follows a multivariate normal distribution:

$$p(\boldsymbol{f}|C_k) = (2\pi)^{-d/2}\,|\boldsymbol{\Sigma}|^{-1/2} exp\,((-\frac{1}{2}(\boldsymbol{f}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{f}-\boldsymbol{\mu})))$$

where $\boldsymbol{\Sigma}$ is the covariance matrix of the training data belonging to C_k, $\boldsymbol{\mu}$ its mean and d the length of $\boldsymbol{f}$. Bayes theory is then implemented to calculate $p(C_k|\boldsymbol{f})$:

$$p(C_k|\boldsymbol{f}) = \frac{p(\boldsymbol{f}|C_k)p(C_k)}{p(\boldsymbol{f})} = \frac{p(\boldsymbol{f}|C_k)}{p(\boldsymbol{f})}\cdot\frac{Nc_k}{N_{train}}$$

where $p(\boldsymbol{f}|C_k)$ is assumed to follow a multivariate normal distribution, $p(C_k)$ is the prior probability of class C_k, N_{C_k} is the number of feature vectors in class k within the training set and $p(\boldsymbol{f})$ is independent of the class. The covariance matrix $\boldsymbol{\Sigma}$ is singular since the number of samples is much lower than the number of dimensions considered. To overcome this issue, PCA-LDA was implemented to perform dimensionality reduction on the data, which thus ensured the non-singularity of $\boldsymbol{\Sigma}$. The Bayes classifier can be visualised by fitting multivariate Gaussian distributions on the data, after PCA-LDA, and plotting their contour lines, as shown in Figure 2 for M_{LDA} = 2. This clearly shows a separation in 8 classes, one for each possible angle. Figure 7 shows the results obtained over 50 iterations using the Bayes classifier with optimal M_{PCA} (determined empirically using 20 iterations in Appendix A.7).

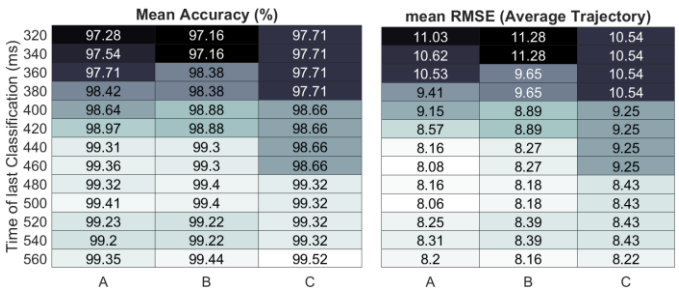


Figure 7 - Accuracy and RMSE (Average Trajectory) using Bayes Classifier. A: δt=20, MPCA=65. B: δt=40, MPCA=69. C: δt=80, MPCA=31. (MLDA=7 and s_P=0.7 in all cases).

The best results for each δt were very close (8.20 for δt=20ms, 8.16 for δt=40ms, 8.22 for δt=80ms). As such, while the best results occurred for δt=40ms, a bin size of 80ms could also be used (resulting in smaller feature vectors and lower computational complexity). Thus, the scheme ultimately used with this method had M_{PCA}=31 and the label was updated at t=320, 400, 480 and 560ms (δt=80ms) (RMSE using average trajectory: mean = 8.22, std = 0.88; over 50 iterations). Further improvements on this method involve fitting the data on different distributions or using hidden Markov models to determine $p(\boldsymbol{f}|C_k)$, as well as optimizing M_{LDA}.

VIII. Majority Voting

In all classifiers considered (k-NN, SVM and Bayesian) δt=80ms was chosen as the optimal bin size. The output of the three classifiers was combined in a majority voting scheme where the final class label was predicted based on the most frequent class label of all methods. This is the primary reason that three classifier methods were chosen. The final class label $\hat{C}$ was obtained at 320ms and updated every 80ms by: $\hat{C} = mode\{\hat{C}_{KNN}, \hat{C}_{SVM}, \hat{C}_{Bayes}\}$. In cases where all methods produce different classification outputs (relative frequency of each predicted class is equal to one), then the final class label was set to the output of the SVM classifier, which produced the highest accuracy and lowest RMSE from all classifier methods.

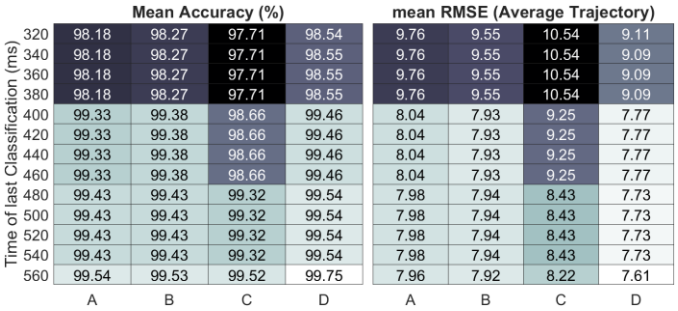


Figure 8 - Accuracy and RMSE (Average Trajectory). A: Nearest Centroid, classifier with M_{PCA}=170. B: SVM classifier with c=1, σ=0.07 and M_{PCA}=180. C: Bayes classifier with M_{PCA}=31. D: Majority Voting scheme. (In all cases M_{LDA}=7, δt=80 and statistics were obtained using 50 iterations and s_P=0.7)

As seen in Figure 8, the majority voting scheme was an improvement over any of the three methods taken individually (peak accuracy was 99.75% at 560ms and RMSE using average trajectory: mean = 7.61, std = 0.82; over 50 iterations).

IX. Principal Component Regression

The x and y hand positions can be predicted by implementing a linear regression model for each angle at each time of interest (t=320ms to t=560ms in steps of 20ms), which aims to estimate the regression coefficient $\boldsymbol{B}$ such that:

$$\boldsymbol{Y} = \boldsymbol{XB} + \boldsymbol{\varepsilon}$$

where $\mathbf{Y}$ corresponds to the centred x and y hand positions at time t (for each class C_k: $\boldsymbol{Y} = \boldsymbol{H}^{C_k}(t) - \bar{\boldsymbol{H}}^{C_k}(t)$), $\mathbf{X}$ is the centred input neural spikes at time t (for each class C_k, $\mathbf{X}$ is constructed by setting each row to a feature vector ($\boldsymbol{f}^T(t)$ or $\tilde{\boldsymbol{f}}^T(t)$) and performing $\boldsymbol{X} \rightarrow \boldsymbol{X} - \bar{\boldsymbol{X}}$) and $\boldsymbol{\varepsilon} \sim N(0, \sigma^2)$. Such problems are often tackled using Ordinary Least Square (OLS) estimation:

$$\hat{\boldsymbol{B}}_{OLS} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}(\boldsymbol{X}^T\boldsymbol{Y})$$

Since the number of dimensions is much higher than the number of samples, $\boldsymbol{X}^T\boldsymbol{X}$ is singular, so OLS solution is intractable. Additionally, noise in $\boldsymbol{X}$ may introduce spurious correlations in the calculation of the regression coefficients. To avoid both these problems, Singular Value Decomposition (SVD) was used to calculate the r largest principal components and thus reduce the dimensionality of the data. Specifically, the SVD of $\boldsymbol{X}$ is given by $\boldsymbol{X} = \boldsymbol{U\Sigma V}^T$, where columns of $\boldsymbol{U}$ are the left singular

vectors, rows of $\boldsymbol{V^T}$ are the right singular vectors and $\boldsymbol{\Sigma}$ is a diagonal matrix whose entries are the singular values of matrix $\boldsymbol{X^TX}$. This method is called Principal Component Regression (PCR) [3] which is based upon least square regression and principal component analysis (PCA). PCR solution ($\widehat{\boldsymbol{B}}_{PCR}$) is obtained by:

$$\widehat{\boldsymbol{B}}_{PCR} = \boldsymbol{V}_{1:r}(\boldsymbol{\Sigma}_{1:r})^{-1}(\boldsymbol{U}^T{}_{1:r}\,\boldsymbol{Y})$$

where the subscript $(1{:}r)$ denotes that the r largest singular values are used in each of $\boldsymbol{U}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{V}$. Then for each testing feature vector $\boldsymbol{x}$(t) fed into the algorithm at time t, an angle $\hat{C}_k$ was predicted and the appropriate $\widehat{\boldsymbol{B}}_{PCR}$ (i.e. the weights corresponding to the predicted angle at time t) were used to estimate the hand position by:

$$\widehat{\boldsymbol{Y}}(t) = \boldsymbol{x}(t)^T\widehat{\boldsymbol{B}}_{PCR}\big(t,\hat{C}_k\big) + \bar{\boldsymbol{H}}_{\boldsymbol{train}}^{\hat{C}_k}(t)$$

Note that for all t > 560ms, $\widehat{\boldsymbol{Y}}(t)$=$\widehat{\boldsymbol{Y}}(560)$ (i.e. the last estimation is retained). Choosing the optimal number of principal components was essential to perform an accurate regression model both in terms of RMSE and of computational complexity. The r was optimized up to $(N_{C_{train}} - 1)$, which is the maximum rank of $\boldsymbol{X^TX}$, as the training data has $N_{C_{train}}$ trials for each angle. This optimization is performed to minimize the RMSE error given perfect angle classification. Figure 9 shows the RMSE (50 iterations, s_p=0.7) of PCR for r up to 69 principal components. When comparing the results with the average trajectory method, PCR performed better for all r. The RMSE decreases with larger number of components, which suggests that using r=$(N_{C_{train}}$ - 1) principal components does not result in overfitting.

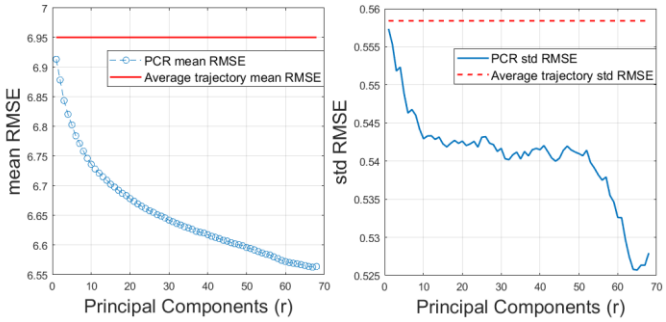


Figure 9 - RMSE plot against increasing number of principal components (r) up to 69, against the average trajectory benchmark method, with s_p=0.7.

Similar trends were observed for different splits (s_p=0.5, 0.6, 0.8, 0.9), as shown in Appendix A.8. This means that for dataset $\boldsymbol{D}$ the maximum number of principal components was needed to capture the information within the data, i.e. $r = (N_{C_{train}} - 1)$ should be taken for all s_p. On the other hand, the first feature vector representation ($\tilde{\boldsymbol{f}}$), which spans a lower dimensionality and whose data is more likely to be correlated showed the lowest RMSE at r=10 (Appendix A.9). To better comprehend this behavior, the normalised eigenvalues corresponding to each angle in the whole dataset for both representations were plotted (Appendix A.10). From the plots in Appendix A.10, it can be seen that for $\tilde{\boldsymbol{f}}$ not all bases were needed to fully capture the information in $\boldsymbol{X^TX}$ ($\sum_{i=1}^{70}\lambda_i / \sum_{i=1}^{99}\lambda_i \approx 0.994$) whereas for $\boldsymbol{f}$, all the principle components (99 in the case of the whole dataset) were needed ($\sum_{i=1}^{98}\lambda_i / \sum_{i=1}^{99}\lambda_i \approx 0.994$).

X. NON-LINEAR LEAST MEAN SQUARES (N-LMS)

Alternatively, to find a nonlinear relationship between the feature vectors and hand position (for each angle at different times t as in Section IX), the nonlinear LMS algorithm can be implemented. This algorithm allows online learning and has increased expressive power and generalization capability relative to linear models. These properties are achieved by introducing a non-linearity to the output of the standard LMS algorithm. A tanh activation function was used, where the hand position is estimated from $\hat{h} = tanh(\boldsymbol{w}^T\boldsymbol{k})$, where $\hat{h}$ represents x or y coordinate estimates in each model, $\boldsymbol{w}$ is the vector containing the LMS weights and $\boldsymbol{k} = [\,\boldsymbol{f^T}\quad 1\,]^{\boldsymbol{T}}$ to account for bias in the inputs. The hand positions were normalized to [-1,1] in order to match the range of the tanh function. The learning rule of the LMS weights is based on stochastic gradient descent:

$$\boldsymbol{w}(n{+}1){=}\boldsymbol{w}(n){+}\mu\big(h(n) - \hat{h}(n)\big)\left(1 - tanh^2\big(\boldsymbol{w}^T(n)\boldsymbol{k}(n)\big)\right)\boldsymbol{k}(n)$$

where μ is the learning rate and h is the true x or y hand position. Since the sample size was insufficient to ensure convergence, the LMS weights were pre-trained by overfitting on 33% of $\boldsymbol{D}_{\boldsymbol{train}}$ (over 500 epochs). As in Section IX, for all t > 560ms, $\widehat{\boldsymbol{Y}}(t)$=$\widehat{\boldsymbol{Y}}(560)$. For optimal μ=0.015 (Appendix A.11), given perfect angle classification, mean RMSE is 6.78 with std=0.52 (50 iterations, s_P=0.7), which is inferior to results obtained using PCR both in terms of RMSE and computational time.

Different activation functions and use of an adaptive learning rate could improve performance of the N-LMS. Additionally, adaptive scaling of the activation function could be used to mitigate normalisation issues. Finally, perceptron models could be connected in a deep neural network to perform regression.

XI. CONCLUSION

To decode the neural spikes of a monkey in order to obtain an estimate of the x and y hand positions, Majority voting was used to classify the angle every 80ms, as described in Section VIII, with optimal hyperparameters derived throughout the report. At each time t, the weights $\widehat{\boldsymbol{B}}_{PCR}$ corresponding to the predicted angle $\hat{C}_k$ were used to estimate the hand position using PCR with $r = (N_{C_{train}} - 1)$. Results for different s_P are shown below.

Table 1. Mean RMSE and Std vs s_P (50 iterations)

s_p	0.5	0.6	0.7	0.8	0.9
Mean RMSE	7.96	7.73	7.37	7.05	6.85
Std RMSE	0.57	0.65	0.67	0.99	1.04

Predicted trajectories for a single run of the algorithm (s_P=0.7) are shown in Appendix A.0. The mean computational time for training the algorithm on the whole dataset $\boldsymbol{D}$ was 26.05s (50 iterations). The mean time needed to predict a single trajectory, after training on the whole dataset, was 0.15s (Appendix A.12).

Averaging methods employed in this report lose some temporal aspects of the neural dynamics. Thus, in future iterations different methods for processing the neural data could be used to better capture the causality and time dependency within the spike trains, such as the van Rossum distance metric [4]. Another improvement would be to use the tuning curves to select optimal neurons both for classification and regression.

XII. AUTHORS CONTRIBUTIONS

GG and **CH**: Features $\boldsymbol{f}$, Tuning curves/Population decoding, PCA-LDA, k-NN, SVM, Majority Voting. **AG** and **EO**: Features $\tilde{\boldsymbol{f}}$, Average Trajectory, Bayesian, PCR, N-LMS.

XIII. REFERENCES

[1] Rao, R. P. N. (2013). "Signal Processing," In: *Brain Computer Interfacing: An Introduction*. Cambridge: Cambridge University Press, pp. 109-148

[2] Keerthi, S. and Lin, C. (2003). *Asymptotic Behaviors of Support Vector Machines with Gaussian Kernel*. Neural Computation, 15(7), pp.1667-1689.

[3] Jolliffe I.T. (2002). "Principal Components in Regression Analysis". In: *Principal Component Analysis*. Springer Series in Statistics. Springer, New York, NY, pp. 167-198

[4] Van Rossum M. (2001). *"A Novel Spike Distance"*. Neural Computation 13, pp. 751-763

XIV. APPENDIX

Appendix A.0

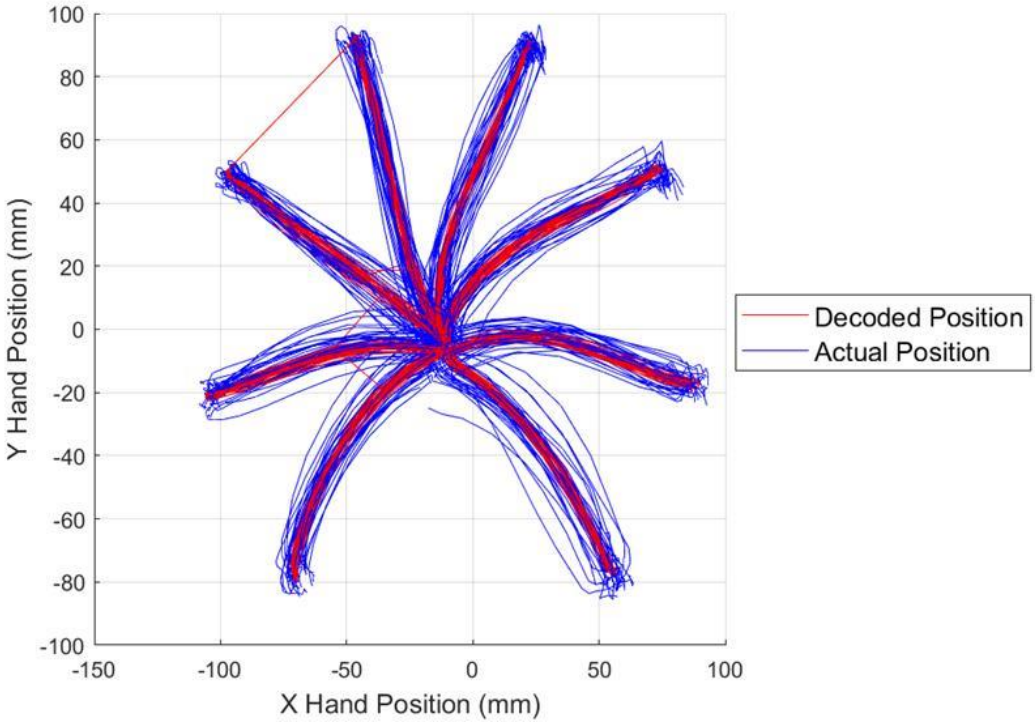


Figure 10 – Predicted and True trajectories for a single run (s_P=0.7) for all angles. Note that predicted angle updates every 80ms until t=560ms, which accounts for the sudden changes in direction in a few of the trajectories. These changes in predicted angle only occur for neighboring directions.

Appendix A.1

S(t) $\in R^{98\times t}$ is a matrix containing all prior neural activity until time t. The matrix $\boldsymbol{V}(t) \in R^{98\times\left\lfloor\frac{t}{\delta t}\right\rfloor}$ was calculated by splitting **S**(t) into bins δt and determining the firing rate within each bin: $V_{i,j}(t) = \frac{1}{\delta t}\sum_{\tau=1+\delta t(j-1)}^{j\ \delta t} S_i^n{}_\tau$.

$\boldsymbol{f}(t) \in R^{98\left\lfloor\frac{t}{\delta t}\right\rfloor\times 1}$ was obtained by: $\boldsymbol{f}(t) = \left[V_{1,1}(t), \dots, V_{98,1}(t), \dots V_{1,\left\lfloor\frac{t}{\delta t}\right\rfloor}(t), \dots, V_{98,\left\lfloor\frac{t}{\delta t}\right\rfloor}(t)\right]^T$, where $\lfloor\cdot\rfloor$ denotes the floor function.

$\tilde{\boldsymbol{f}}(t) \in R^{98\times 1}$ was obtained by: $\tilde{\boldsymbol{f}}(t) = \frac{1}{t}\sum_{\tau=1}^{t}\left[S_{1,\tau}(t), S_{2,\tau}(t), \dots, S_{98,\tau}(t)\right]^T$

Appendix A.2

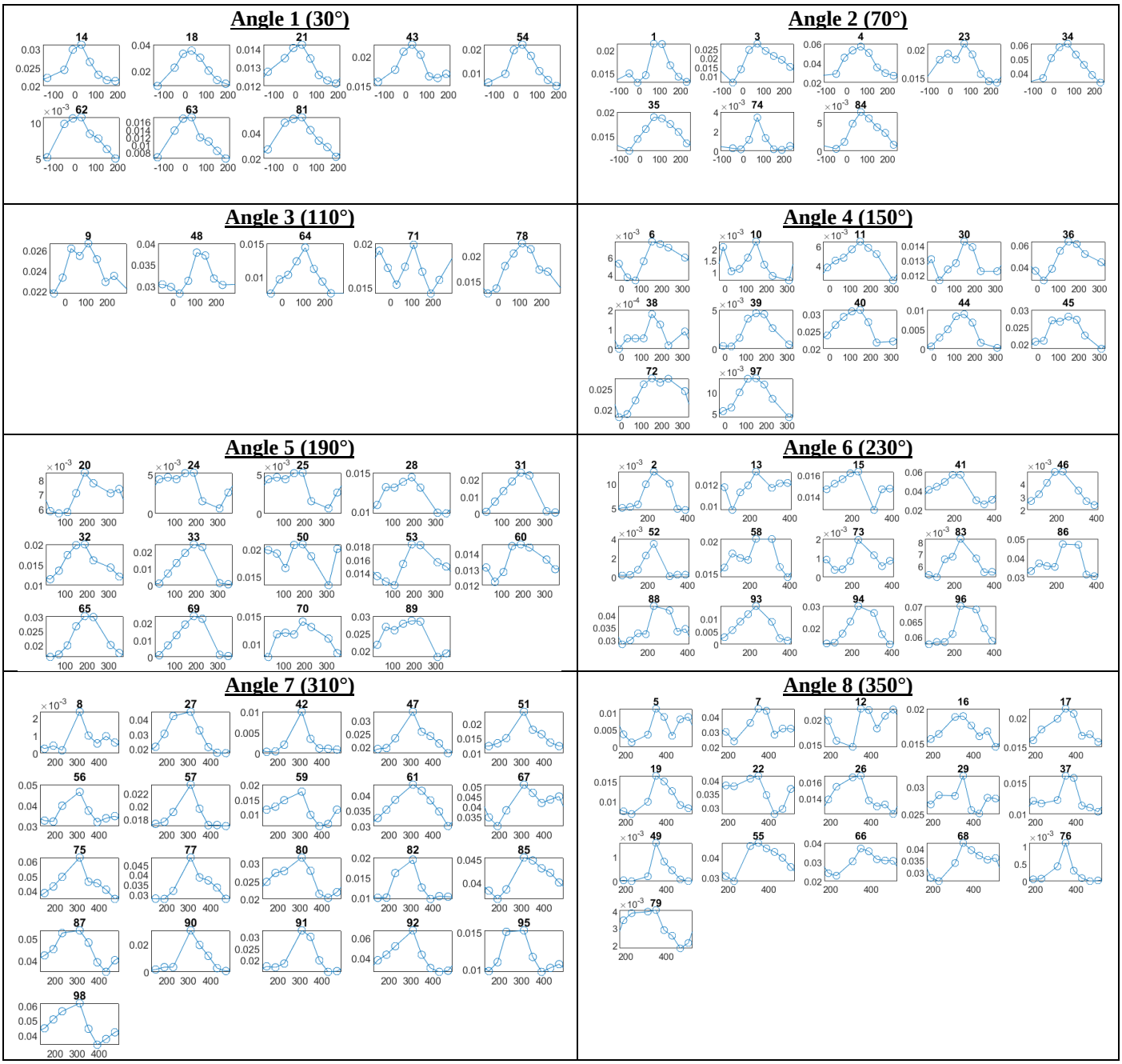


*Figure 11 - Unnormalized Tuning Curves (centered at peak value/preferred direction) for all 98 neurons at time t=560ms, using the all of **D**. x axis represents angle, y axis represents firing rate, title represents neuron index*

To optimize population decoding, the neurons that best represent each preferred direction had to be determined. This was performed by first filtering out the unnormalized tuning curves that presented a large ratio of standard deviation of the peak value through trials over the mean peak value through trials. This metric quantifies the potential error in determining both the peak value and its respective angle (it was found, that setting a threshold of 0.5 filtered out all highly problematic cases). Subsequently, the two best normalized tuning curves for each preferred direction were chosen with respect to the difference between the means of the two largest values, hence quantifying the sharpness of the tuning curve. Only two neurons were chosen per angle since the 110° angle had only 2 neurons associated with it in most splits of the training data. Normalized tuning curves that satisfy the criterion set (std at peak over mean peak value less than 0.5) with the accompanying std/mean metric for the complete dataset ***D*** are plotted below.

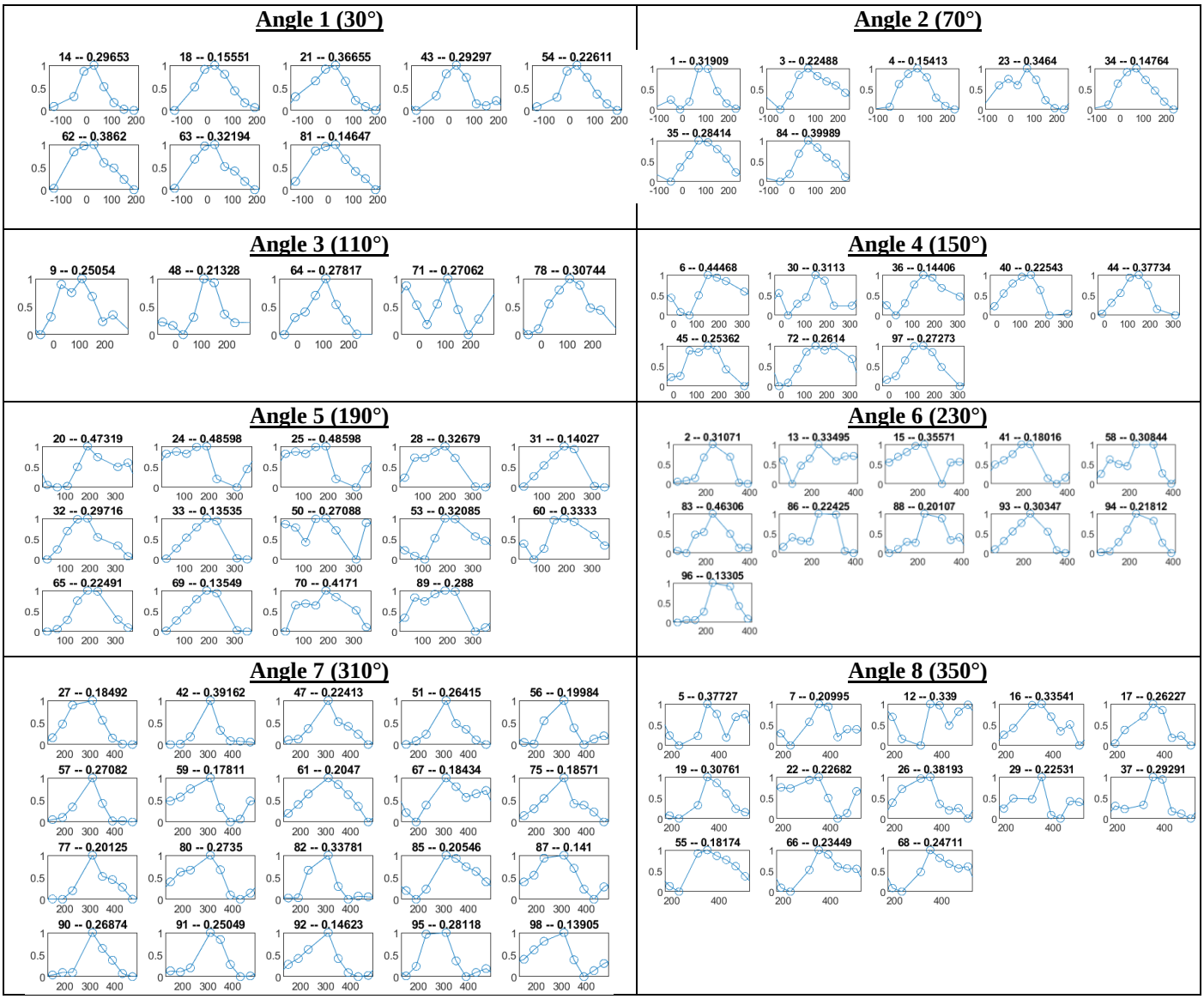


*Figure 12 – Normalized Tuning Curves (centered at peak value/preferred direction) for the 86 neurons satisfying criterion set, at time t=560ms, using the all of **D**. x axis represents angle, y axis represents normalized firing rate (such that maximum is 1 and minimum is 0), title represents neuron index along with the value of the std/mean metric*

The 2 neurons for each angle were chosen automatically at each time (t=320ms to t=560ms in steps of 20ms), based on satisfying the std/mean threshold of 0.5 in the unnormalized tuning curves and maximizing the largest to second-largest distance in the normalized tuning curves. Population decoding results using these neurons over 20 iterations are shown below:

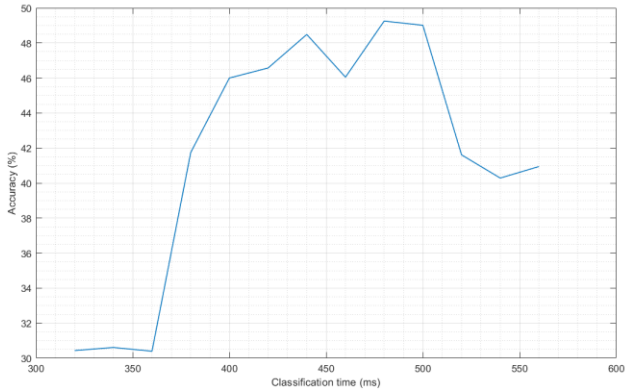


Figure 13 - Mean Classification Accuracy vs Classification time (s_P=0.7, 20 iterations). Peak is 49.25% at 480ms.

Appendix A.3

Linear Discriminant Analysis (LDA) finds the optimal direction to separate data of different classes by rotating the feature space. LDA is applied to the output (the weight matrices) of the PCA technique. The first step is to calculate the mean ($\boldsymbol{m}_i$) of the individual classes as well as the overall mean feature vector ($\widetilde{\boldsymbol{m}}$). Then, the between class scatter matrix ($\mathbf{S}_B$) and the within-class scatter matrix ($\mathbf{S}_W$) are calculated according to the following equations:

$$\boldsymbol{S}_B = \sum_{i=1}^{c} (\boldsymbol{m}_i - \widetilde{\boldsymbol{m}})\ (\boldsymbol{m}_i - \widetilde{\boldsymbol{m}})^T$$

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \sum_{x \in c_i} (\boldsymbol{x} - \boldsymbol{m}_i)\,(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

where $\boldsymbol{x}$ is every feature vector of the training set. Whilst PCA increases data variance both in $\mathbf{S}_B$ and $\mathbf{S}_W$, LDA only increases data variance $\mathbf{S}_B$ and minimises the within class variance $\mathbf{S}_W$. The optimization problem can be described by the following equations:

$$\boldsymbol{W}_{PCA} = argmax_W |\boldsymbol{W}^T \boldsymbol{S}_T \boldsymbol{W}|$$

$$\boldsymbol{W}_{LDA} = argmax_W \left| \frac{\boldsymbol{W}^T \boldsymbol{S}_B \boldsymbol{W}}{\boldsymbol{W}^T \boldsymbol{S}_W \boldsymbol{W}} \right|$$

Where S_T is the total scatter matrix $\boldsymbol{S}_T = \boldsymbol{S}_B + \boldsymbol{S}_W$.
In the case of LDA, to find the direction that separates the data in different classes, we aim to maximise the numerator $\boldsymbol{W}^T \boldsymbol{S}_B \boldsymbol{W}$ and minimising the denominator $\boldsymbol{W}^T \boldsymbol{S}_W \boldsymbol{W}$. PCA is implemented to reduce the dimensionality of the data, keeping the M_{PCA} eigenvectors with the largest eigenvalues, and then performing LDA. The solution to the PCA-LDA algorithm becomes:

$$\left(\boldsymbol{W}_{PCA}{}^T \boldsymbol{S}_W \boldsymbol{W}_{PCA}\right)^{-1} \left(\boldsymbol{W}_{PCA}{}^T \boldsymbol{S}_B \boldsymbol{W}_{PCA}\right) \boldsymbol{W} = \lambda \boldsymbol{W}$$

$\mathbf{W}_{LDA}$ is obtained by keeping the M_{LDA} eigenvectors of $\mathbf{W}$ with the largest eigenvalues. In the end the two methods are combined according to the following transformation:

$$\boldsymbol{W}_{opt}{}^T = \boldsymbol{W}_{LDA}{}^T \boldsymbol{W}_{PCA}{}^T$$

Appendix A.4

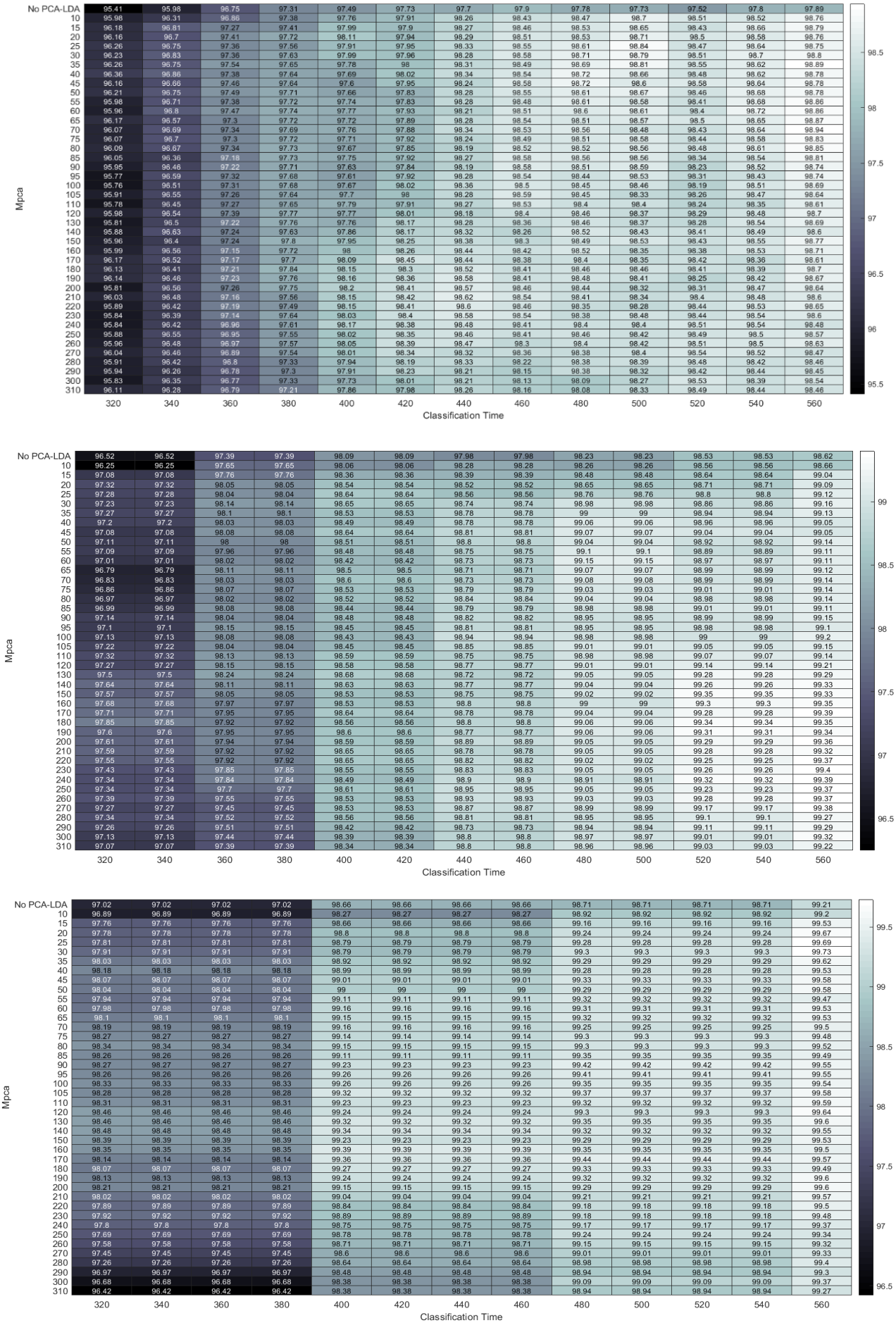
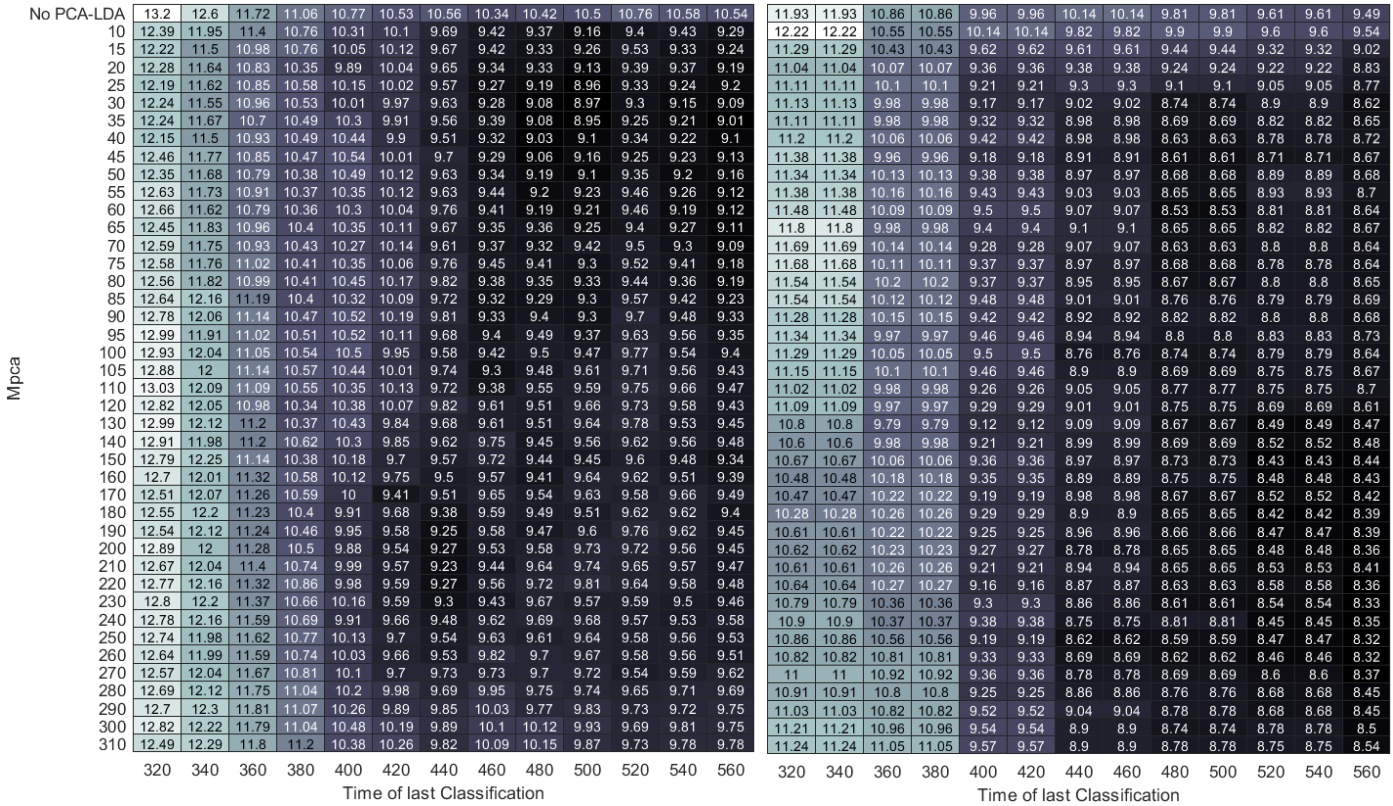


Figure 14. Mean classification accuracy for k-NN over 20 iterations. From top to bottom: δt = 20, 40, 80.



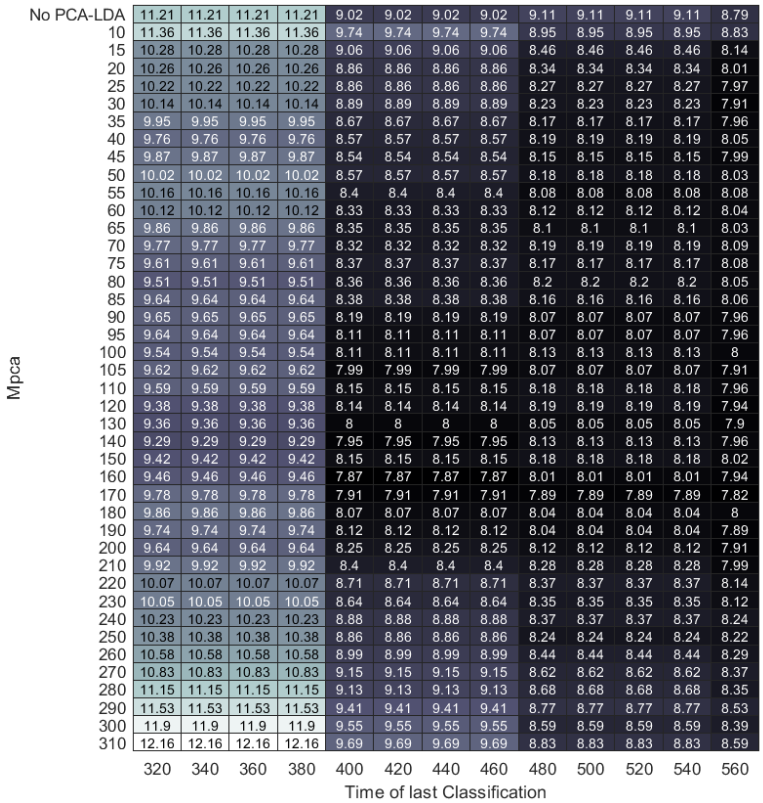


Figure 15. Mean RMSE for k-NN over 20 iterations. Top left: δt = 20ms. Top-right: δt= 40ms. Bottom-left: δt= 80ms.

Minimum for δt=20 occurred at M_{PCA}=35, for δt=40 at M_{PCA}=260, for δt=80 at M_{PCA}=170 with (MLDA=7 in all cases). With δt=80 and M_{PCA}=170 producing best results. Note that within each heatmap, multiple values of M_{PCA} produce results within 0.1 of the minimum.

Appendix A.5

Comparison of Linear vs Gaussian kernel for SVM, using all feature vectors or mean feature vectors. For all cases considered, t=320ms, δt=20ms, s_P=0.7 and 20 iterations were used to obtain the statistics shown.

- When all feature vectors were used, the training input was $70 \times 8 = 560$ feature vectors. The first 2 layers of the decision tree function as shown in Figure 5. However, in the 3rd layer the nearest centroid classifiers were replaced with SVM classifiers (whose training input was $70 \times 2 = 140$ feature vectors). Both the Linear and Gaussian kernels performed much worse than the k-NN classifiers (compare values in Figure below to results in Section V). Hence, this method was abandoned in favour of using the mean feature vectors as inputs (similarly to the case of k-NN in Section V).

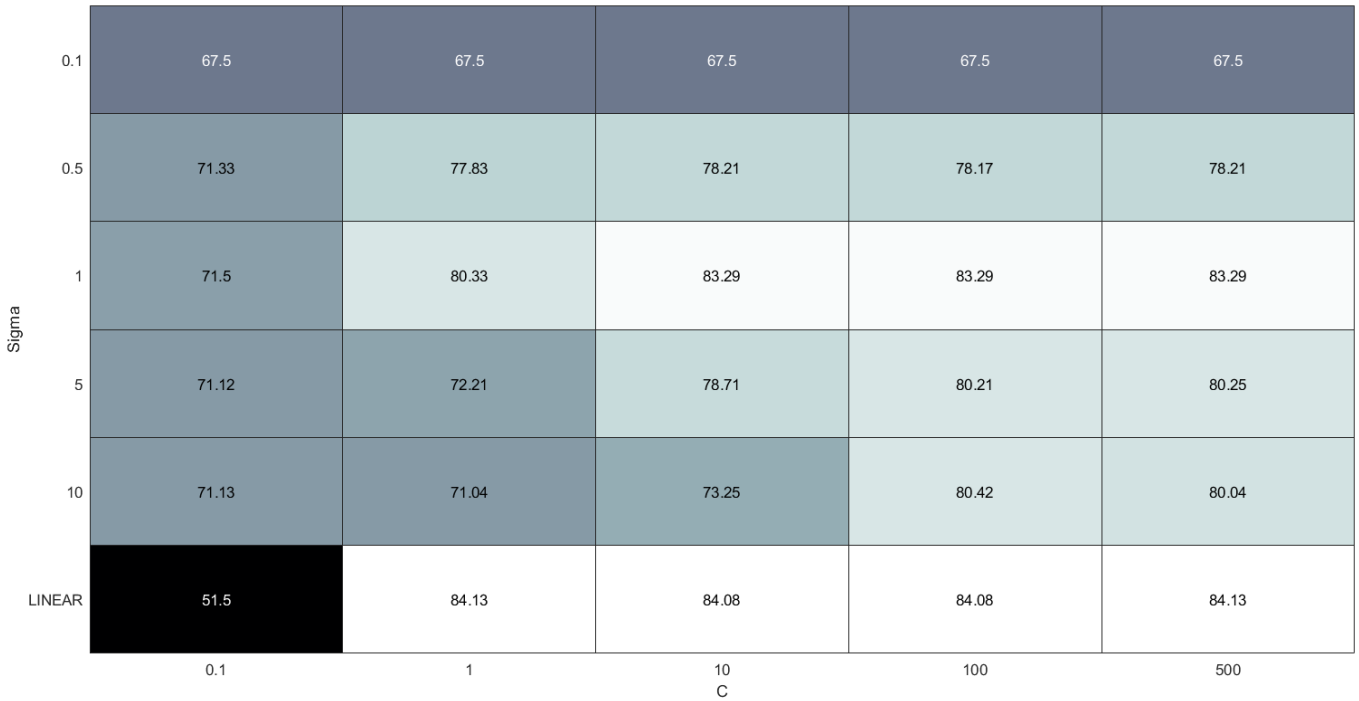


Figure 16 – Comparison of Performance of Linear Kernel SVM vs Gaussian Kernel SVM when all feature vectors are used for different c, σ values. Linear kernel performance is shown on last row (t=320ms, δt=20ms, s_P=0.7 and 20 iterations).

- When only the mean feature vectors were used, the training input was 8 feature vectors. The classifier was structured as shown in Figure 5. The performance for the Gaussian kernel is extensively covered in Section VI as well as in Appendix A.6. For the Linear kernel, the mean accuracy was found to be 72.46% for c=1 (where, as in the Gaussian case for mean vector input, the influence of c on RMSE is minor). Results for both kernels using both types of inputs are tabulated below:

Table 2 – Comparison of all methods considered for SVM (t=320ms, δt=20ms, s_P=0.7 and 20 iterations)

Method	**Mean accuracy (%)**
Linear kernel with all feature vectors (c = 1)	84.13
Gaussian kernel with all feature vectors (c = 10, σ = 1)	83.29
Linear kernel with mean feature vectors (c = 1)	72.46
Gaussian kernel with mean feature vectors (c = 1, σ = 0.1)	95.84

As seen in Table 2, when all feature vectors are used for training, the Linear and Gaussian performances are comparable. However, when the input is converted to the mean feature vector for each angle, the performance of the Gaussian kernel increases and the performance of the Linear kernel decreases.

Appendix A.6

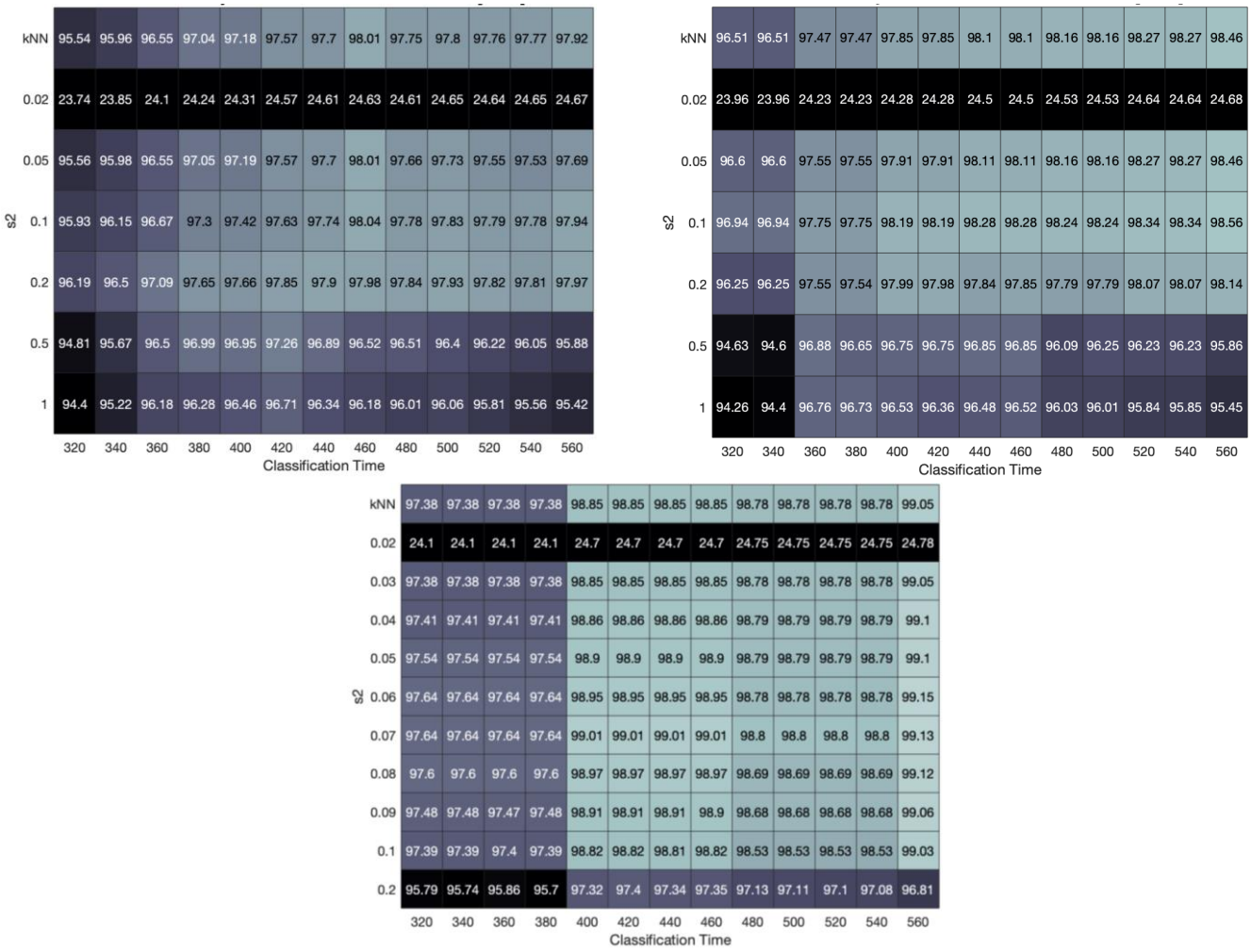


Figure 17 - Mean classification accuracy for SVM over 20 iterations with c=1 for different values of σ (denoted by s_2 in the heatmap y-axis) Top left - δt=20, Top right - δt=40, Bottom δt=80

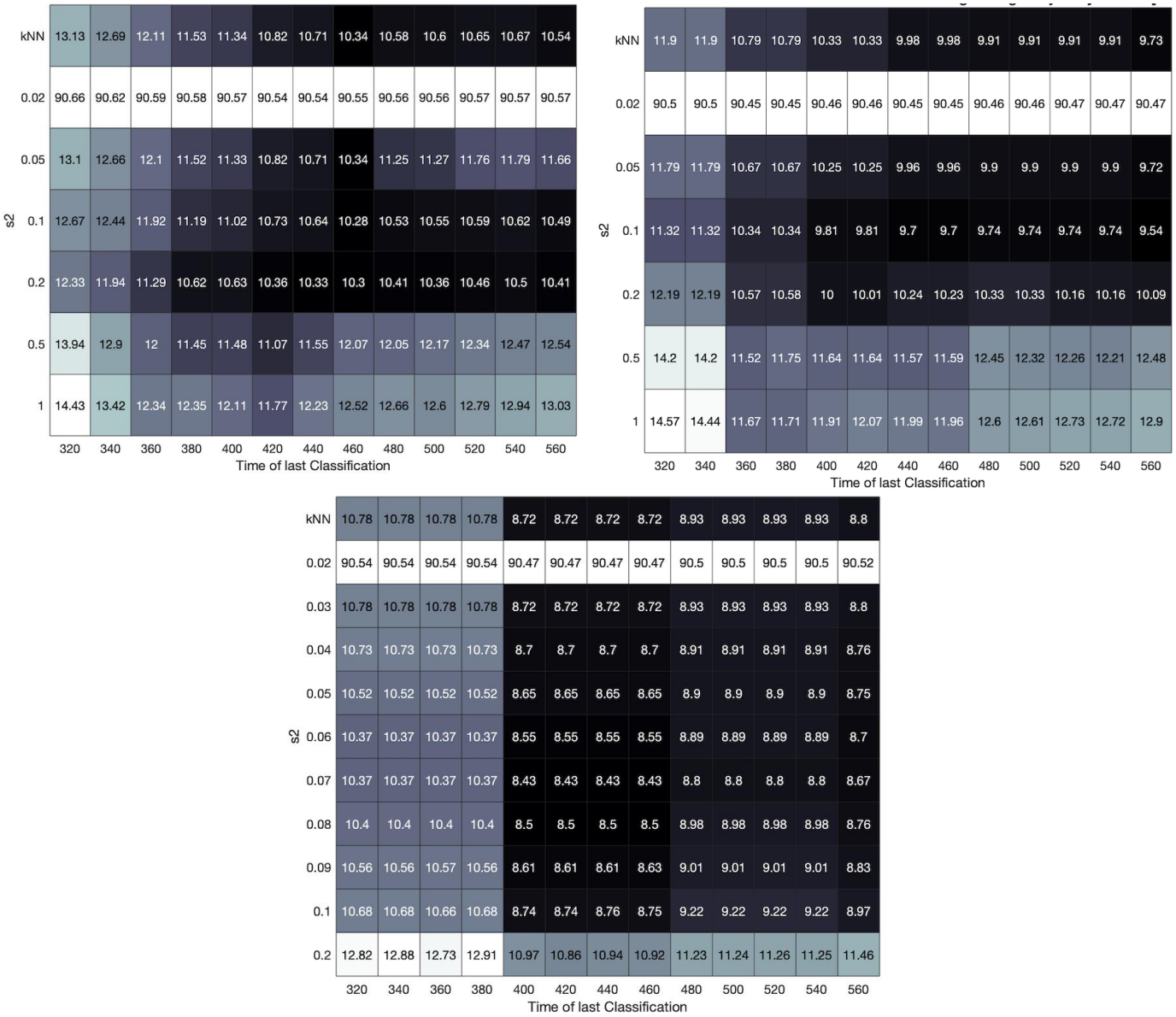


Figure 18 - Mean RMSE for SVM over 20 iterations with c=1 for different values of σ (denoted by s_2 in the heatmap y-axis). Top left - δt=20, Top right - δt=40, Bottom δt=80

Minimum for δt=20 occurred at σ = 0.2, for δt=40 at σ = 0.1 and for δt=80 at σ = 0.07. All values of σ in the [0.05 – 0.1] range produced very similar results. For convenience, at δt=20 the σ chosen was 0.1 (which is acceptable since the results between σ=0.1 and σ = 0.2 vary by only 0.02 in mean RMSE.

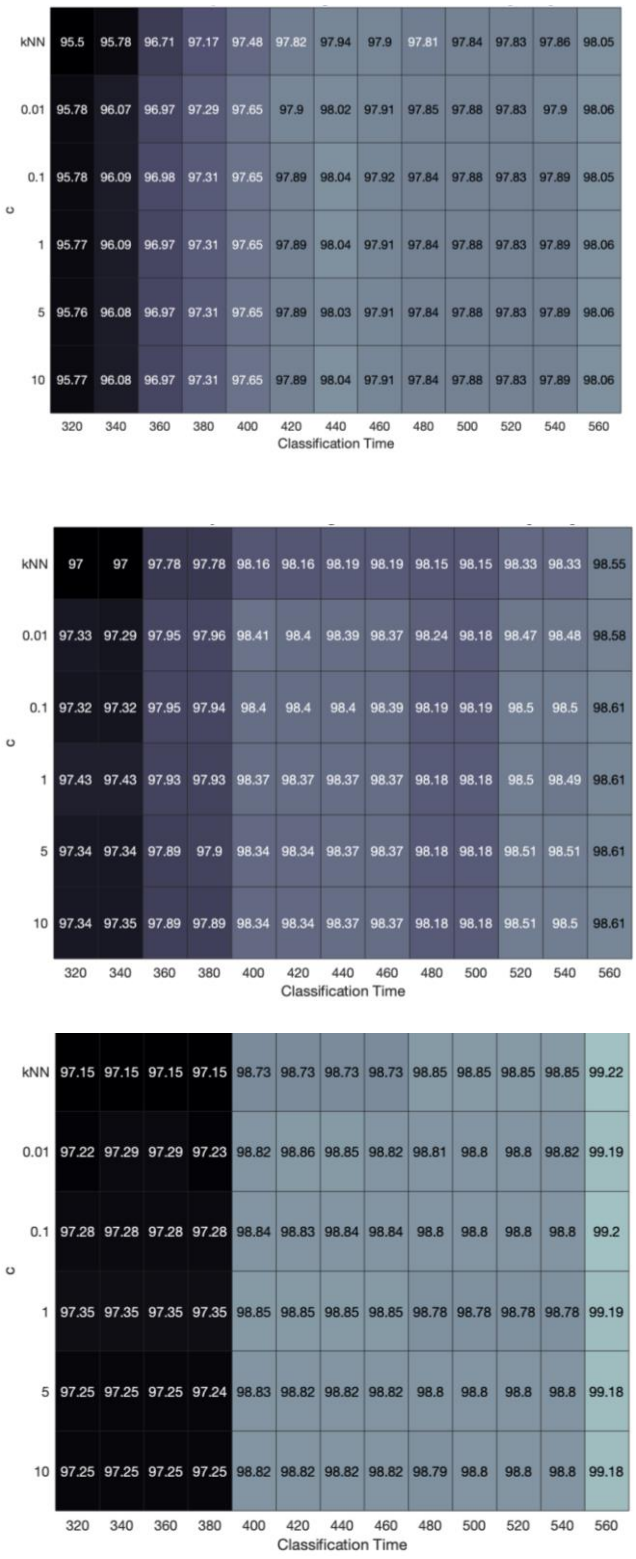


Figure 19 - Mean classification accuracy for SVM over 20 iterations for different values of c. From top to bottom: (δt=20, σ=0.1), (δt=40, σ=0.1), (δt=80, σ=0.07)

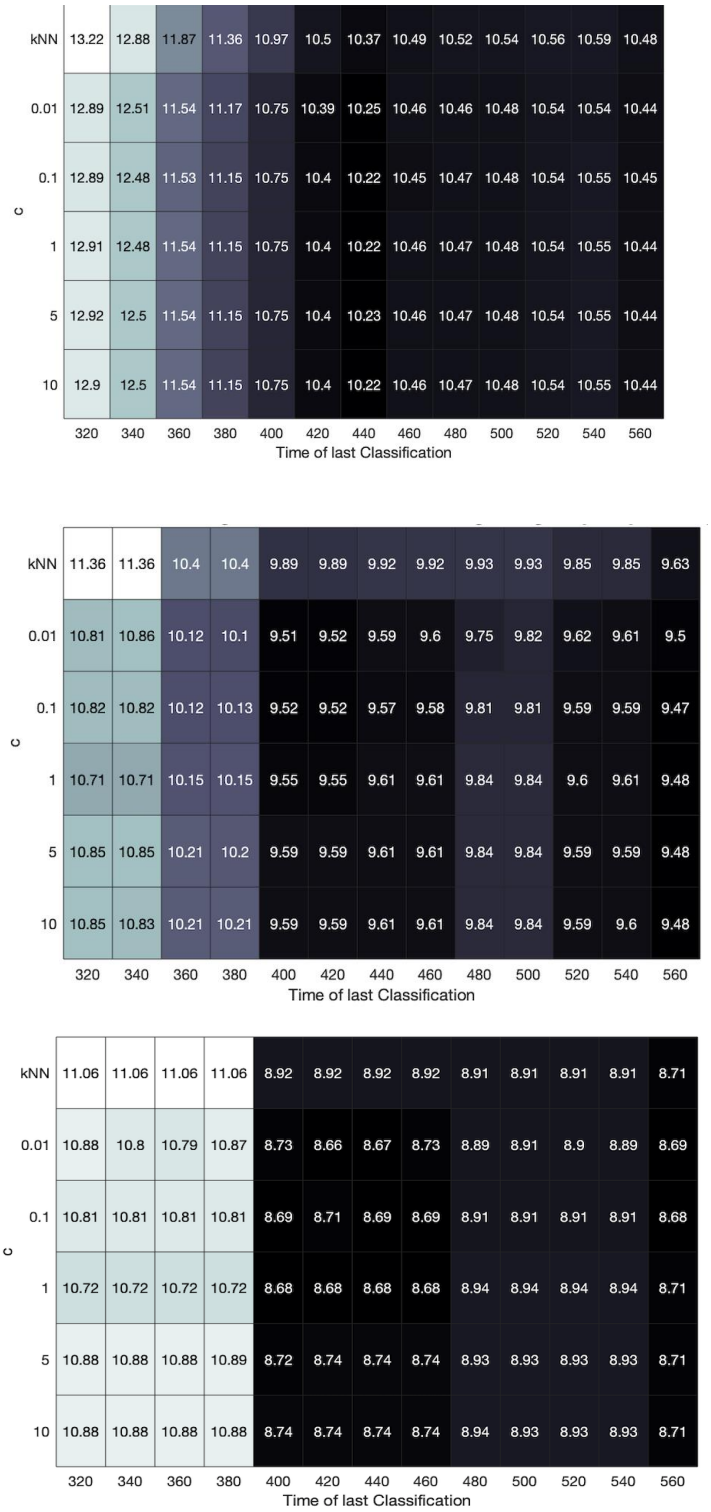


Figure 20 - Mean RMSE for SVM over 20 iterations for different values of c. From top to bottom: (δt=20, σ=0.1), (δt=40, σ=0.1), (δt=80, σ=0.07).

There is negligible variation in results for all values of c from 0.01 to 10. c=1 was chosen for all cases for convenience.

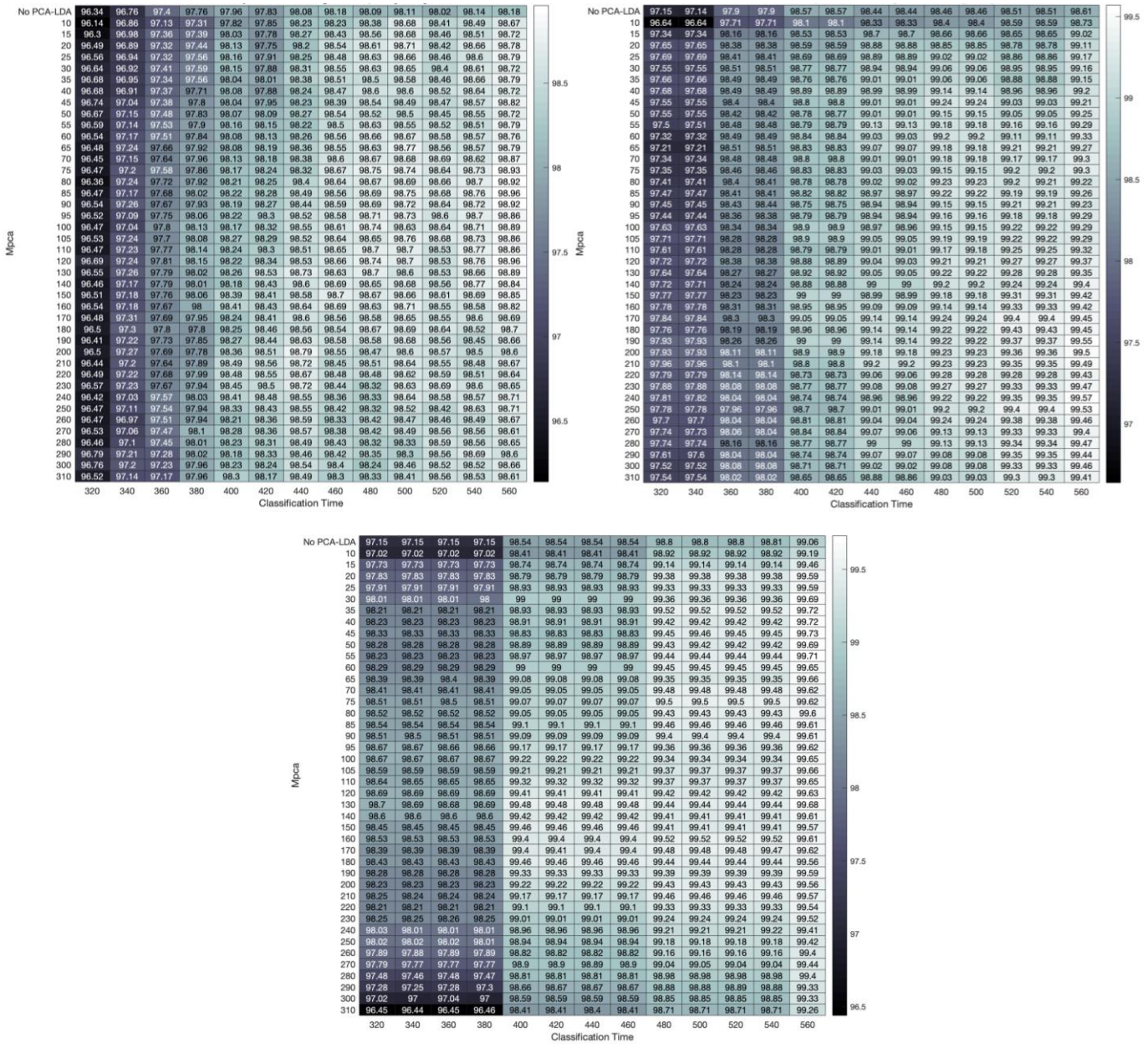
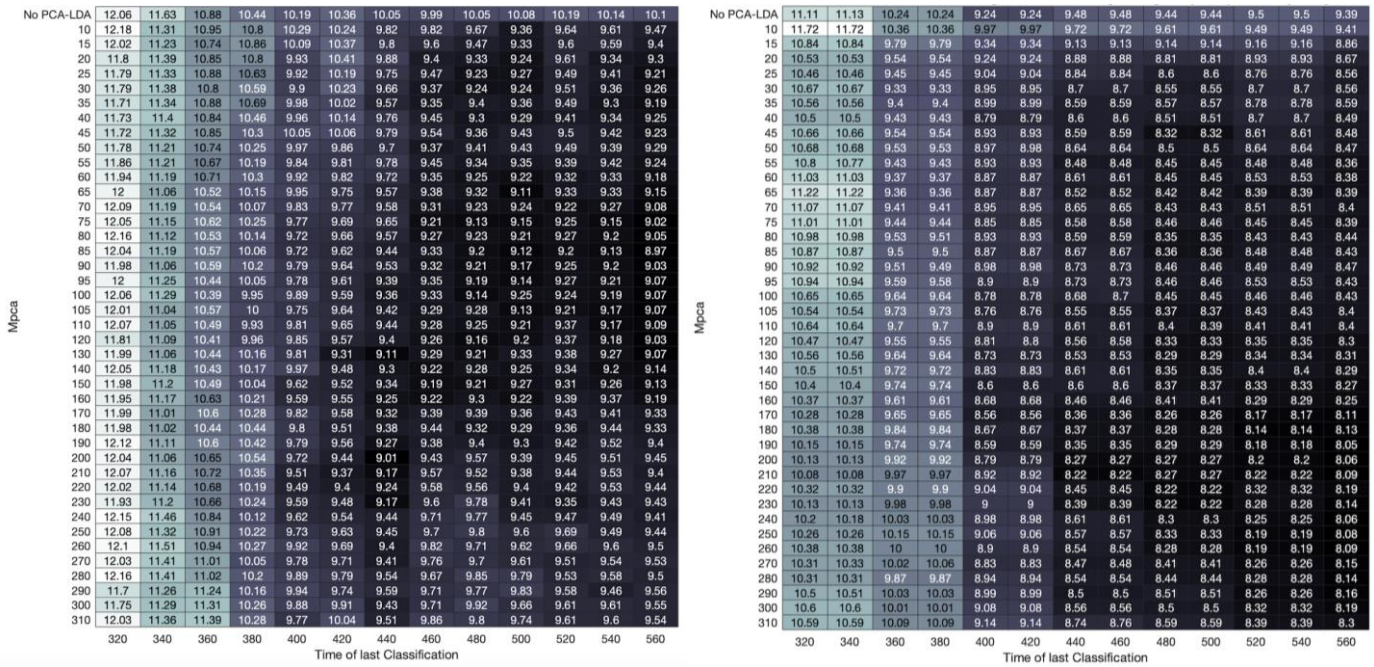


Figure 21 - Mean classification accuracy for SVM over 20 iterations with σ=0.1, c = 1, M_{LDA} = 7, for different values of M_{PCA}. Left - (δt=20, σ=0.1), Right - (δt=40, σ=0.1), Bottom - (δt=80, σ=0.07)



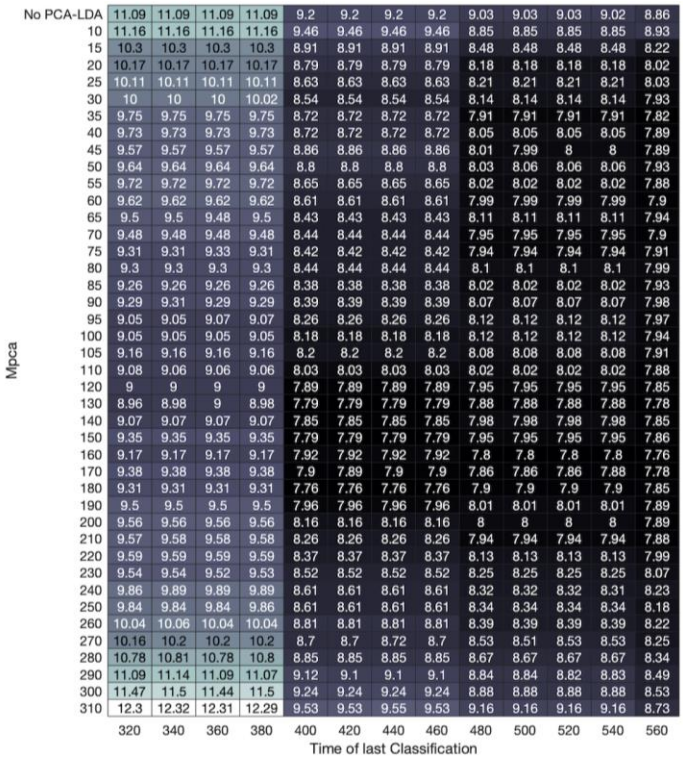


Figure 22 - Mean RMSE for SVM over 20 iterations with c = 1, M_{LDA} = 7, for different values of M_{PCA}
Left - (δt=20, σ=0.1), Right - (δt=40, σ=0.1), Bottom - (δt=80, σ=0.07)

Optimal hyperparameters: for δt=20, (σ=0.1, MPCA=85), for δt=40, (σ=0.1, MPCA=190), for δt=80, (σ=0.07, MPCA=180). Note that within each heatmap multiple values of M_{PCA} produce results within 0.1 of the minimum.

Appendix A.7

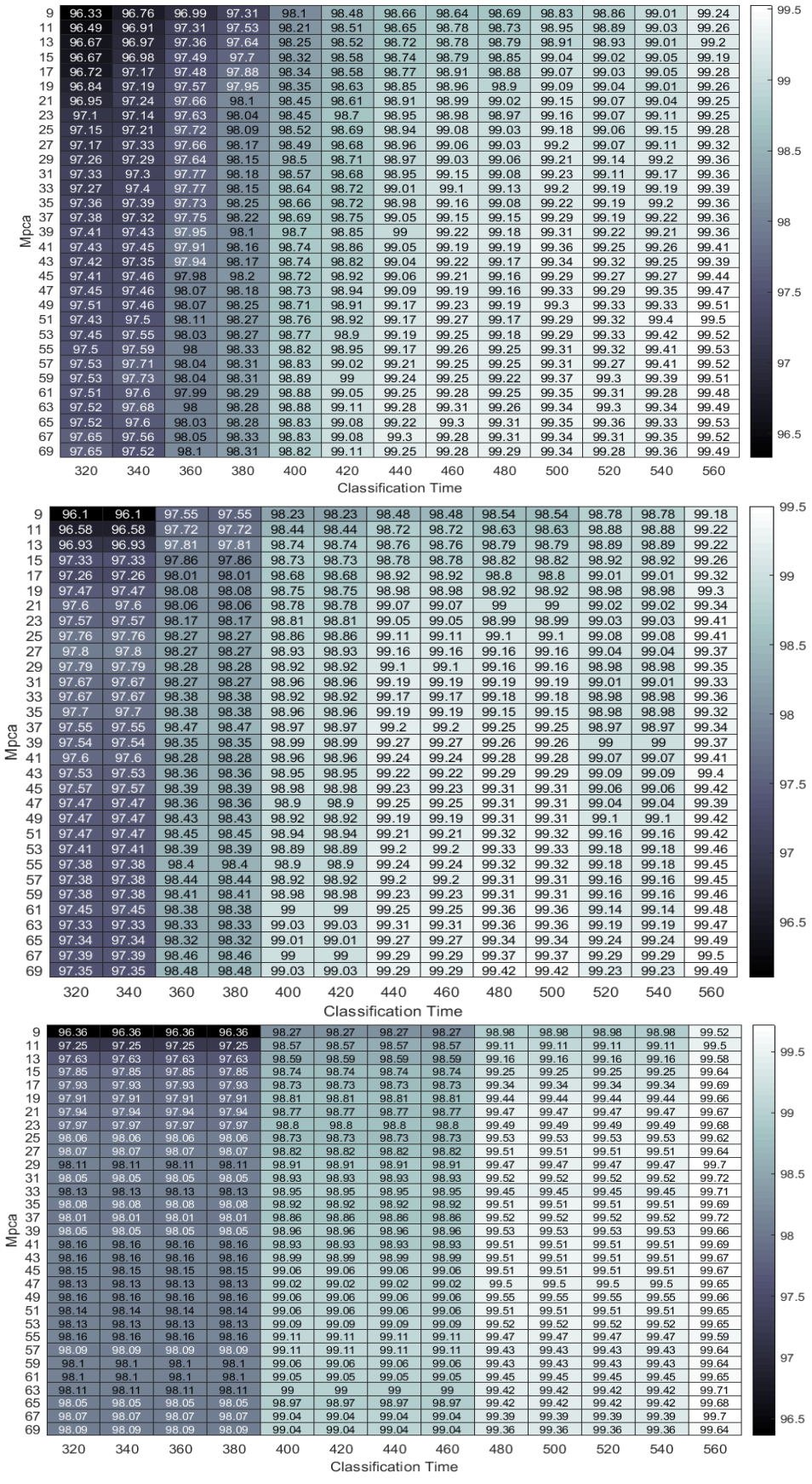


Figure 23. Mean classification accuracy for Bayes over 20 iterations. From top to bottom: δt = 20, 40, 80.

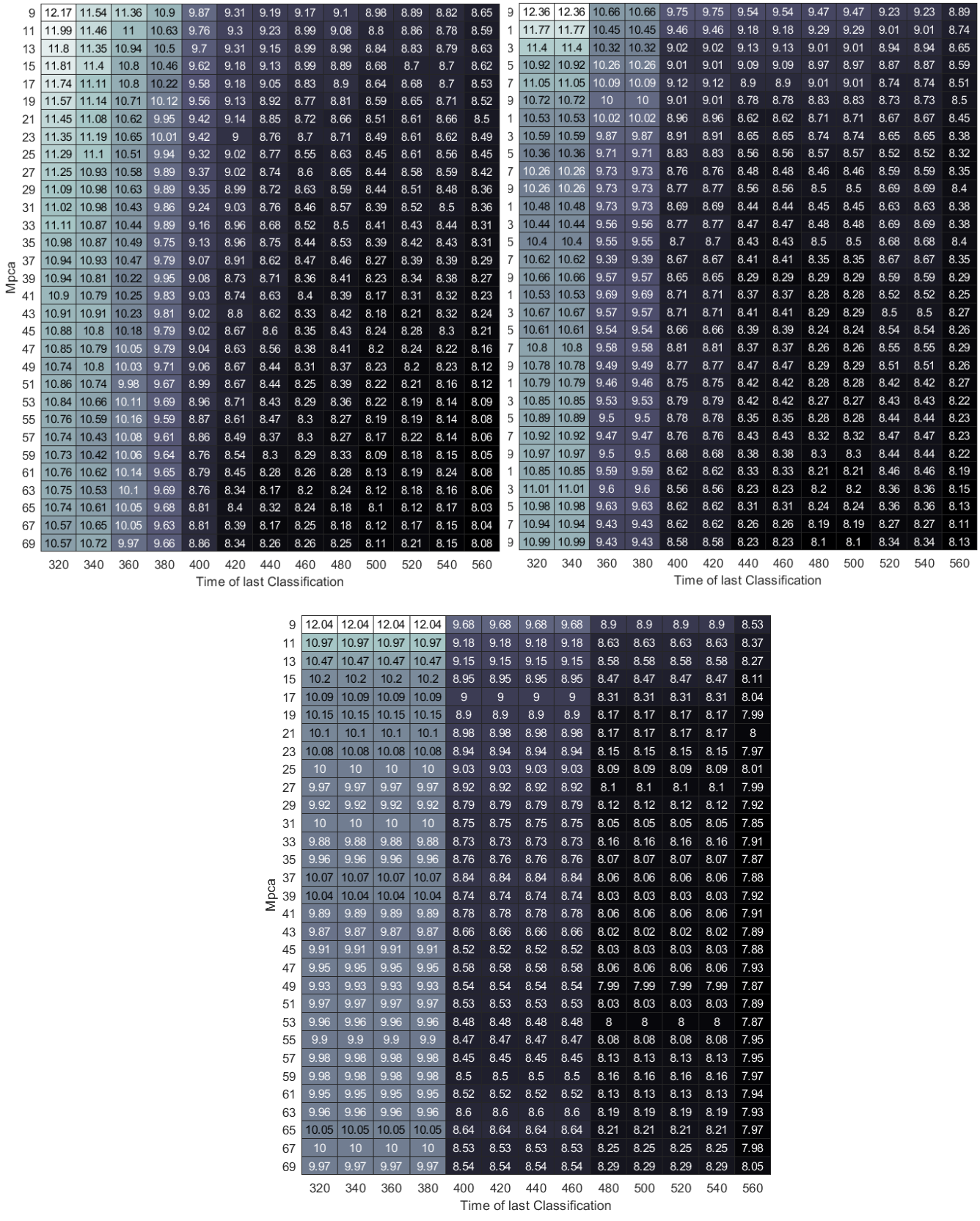


Figure 24. Mean RMSE for Bayes over 20 iterations. Top-left: δt = 20ms. Top-right: δt= 40ms. Bottom: δt= 80ms.

Optimal hyperparameters: for δt=20, MPCA=65, for δt=40, MPCA=69 and for δt=80, MPCA=31. Note that within each heatmap multiple values of M_{PCA} produce results within 0.1 of the minimum.

Appendix A.8

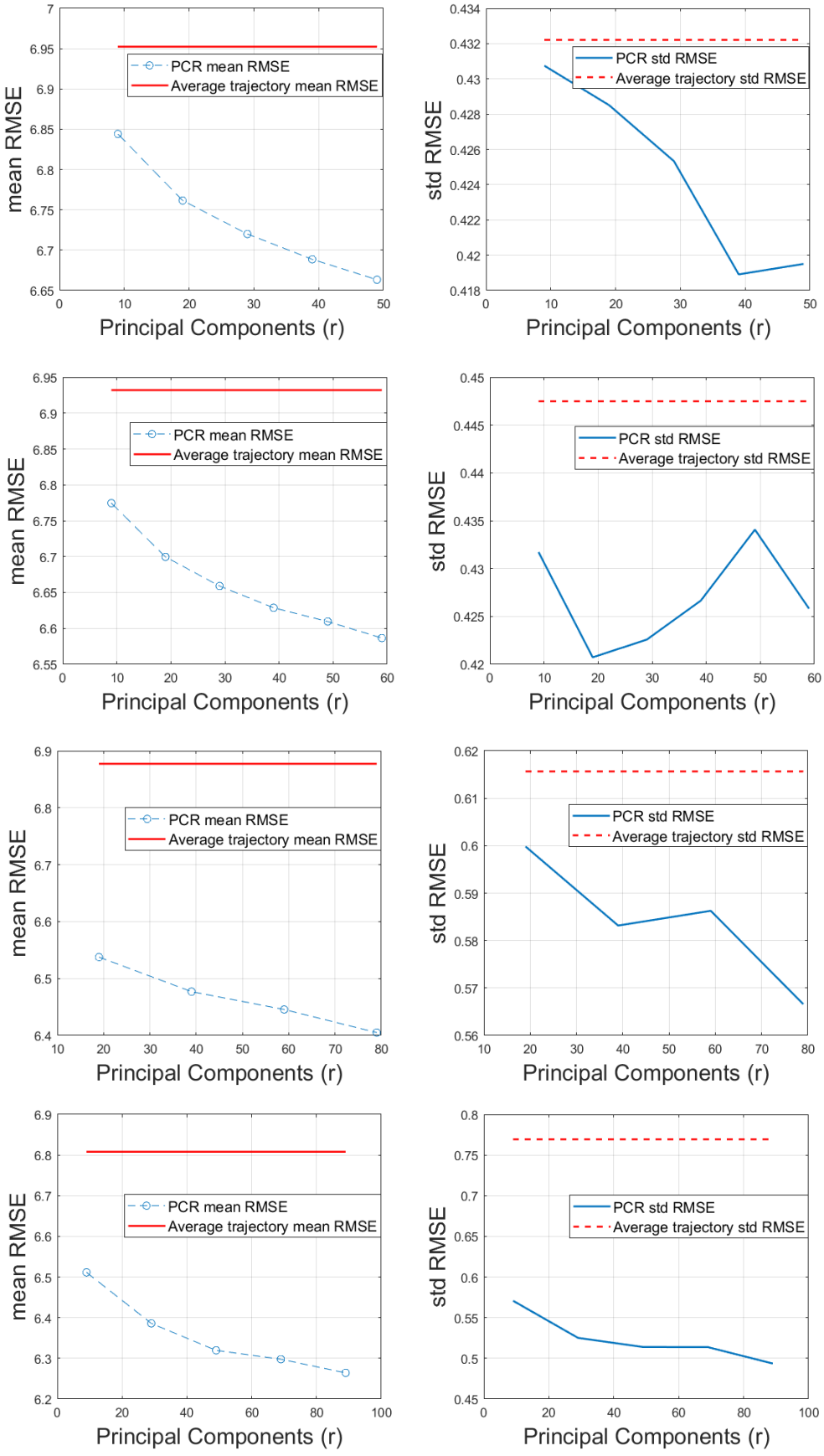


Figure 24. Mean and std RMSE against r principal components and for different s_P. From top to bottom: s_P = 0.5, 0.6, 0.8, 0.9.

The lowest mean RMSE occurs at the maximum number of principle components in all cases. Thus, when using the whole dataset to train the algorithm the maximum number of bases should be used (r = 99).

Appendix A.9

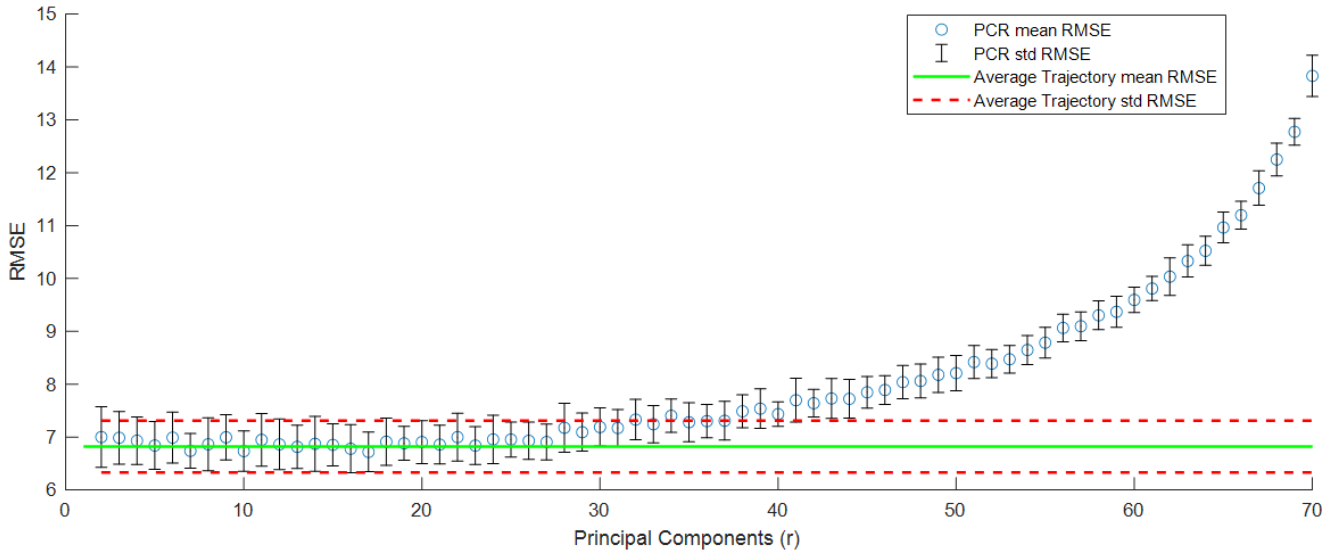


Figure 25 -. Mean and std RMSE with PCR and Average trajectory, against r (20 iterations, s_P=0.7) using Feature vector representation $\tilde{\boldsymbol{f}}$.

The mean RMSE is lowest at r = 10 and increases for larger numbers of principal components. This suggests that the information in the dataset can be expressed using only few principal components. Note that this behaviour is very different to that observed for representation $\boldsymbol{f}$. Additionally, the mean RMSE is higher for all r than the optimal result using $\boldsymbol{f}$, thus the N-LMS and final algorithm were based on representation $\boldsymbol{f}$.

Appendix A.10

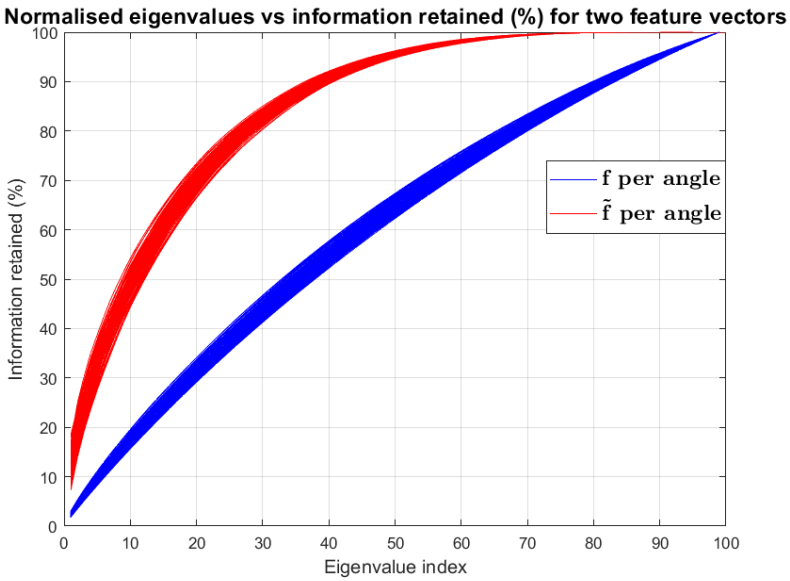


Figure 26. Normalized eigenvalues of the matrix $(\boldsymbol{X}-\bar{\boldsymbol{X}})^T(\boldsymbol{X}-\bar{\boldsymbol{X}})$, with the matrix $\boldsymbol{X}$ as defined in Section IX (but with s_P=1, i.e. considering the whole dataset), against percentage of information retained for both feature vector representations, for each of the 8 angles (angles 1 to 8) and 13 distinct timepoints (t=320ms to 560ms in steps of 20ms). This means that there are 104 red and 104 blue lines plotted within the graph (with all red and all blue lines being virtually indistinguishable from each other)

For representation $\tilde{\boldsymbol{f}}$**:** In all cases considered, the first 70 eigenvalues capture more than 99.5% of the information.
For representation $\boldsymbol{f}$**:** In all cases considered, more than 97 eigenvalues are required to capture more than 99.5% of the information.
Hence, all principal components are needed to successfully express $(\mathbf{X}-\bar{\mathbf{X}})^{\mathrm{T}}(\mathbf{X}-\bar{\mathbf{X}})$**.**

Appendix A.11

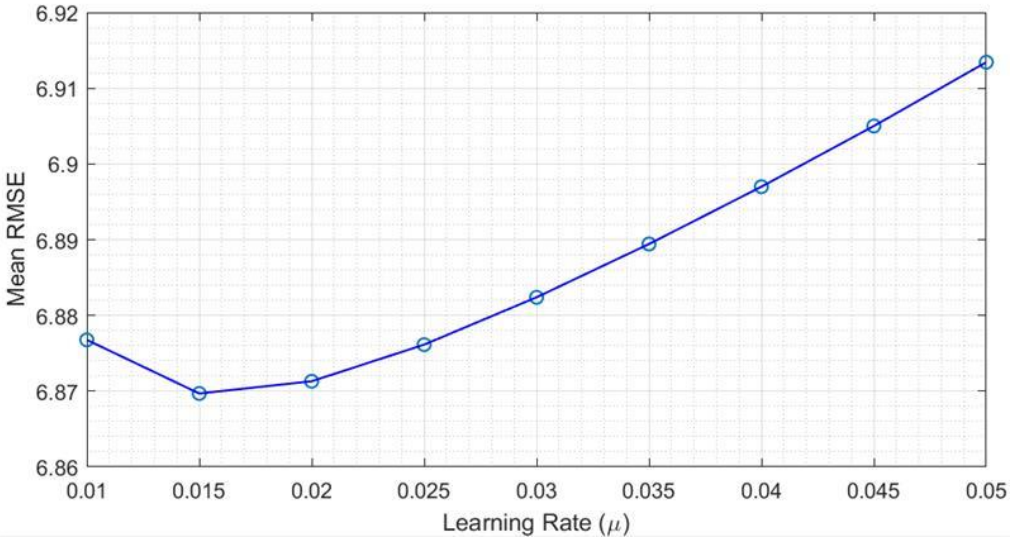


Figure 27 – mean RMSE using N-LMS vs learning rate μ. Angles were perfectly classified, to eliminate misclassification effects on the RMSE. Using s_P=0.7 and over 20 iterations, the minimum mean RMSE occurred at μ=0.015.

One model was created for each time (320ms to 560ms in steps of 20ms) as well as for each of the 8 angles (total $8 \times 13 = 104$ models). To generate these models, the nonlinear LMS learning rule was applied (as specified in Section X) where here *n* represents the trial index (i.e. the algorithm is being looped through the 70 trials that correspond to each time-angle). The same learning rate was used for all models, which may potentially be sub-optimal. Hence, the results could be further improved by optimizing a separate learning rate μ for each model.

Appendix A.12

To assess the time necessary for the complete algorithm, the training time using the whole dataset $\boldsymbol{D}$ was recorded over 50 iterations. (mean = 26.05s, with std = 3.24s). Subsequently, the model produced from training on the whole data was then used to estimate a set of 800 trajectories. Since all the data available was used for training, the same dataset $\boldsymbol{D}$ had to be used for trajectory decoding. This means that the RMSE output would not be representative of the performance of the algorithm, however, the time performance should provide a valid indication of the time per estimate of the algorithm. Dataset $\boldsymbol{D}$ contains 800 trajectories and it was found to require 122s on average to be decoded fully, using the model trained on the whole of $\boldsymbol{D}$ (50 iterations). Thus, the mean time required to decode a single trajectory is approximately 0.153s.

The total runtime of the algorithm can be estimated as $26.05 + 0.153 \times N$ seconds, where N is the number of trajectories decoded (for the validation dataset provided in the competition $N = 82 \times 8 = 656$, and hence the estimated runtime is 126.42s). Note that runtimes vary with system specifications and as such are only meant as an indicator of time complexity/performance.

Appendix A.13

For cross-validation, the set **D** was randomly split into $\mathbf{D}_{\mathbf{train}}$ and $\mathbf{D}_{\mathbf{test}}$ using random permutations of the trial indexes. The sizes of $\mathbf{D}_{\mathbf{train}}$ and $\mathbf{D}_{\mathbf{test}}$ were kept constant (set by the training-to-test ratio s_P). It was also ensured that all classes had the same sample size within each of $\mathbf{D}_{\mathbf{train}}$ and $\mathbf{D}_{\mathbf{test}}$. This process was performed multiple times/iterations and then the mean and std of the accuracy and RMSE results produced over all iterations were reported. This method is defined as Monte-Carlo cross-validation and was preferred over k-fold cross-validation as it explores a larger number of ways of partitioning the data with the drawback of not necessarily including all datapoints in the testing set at least once (as in k-fold). To mitigate this drawback a significant number of iterations are necessary (20 iteration were used for optimization tasks, while 50 were used for any evaluations). Clearly, increasing the number of iterations would yield more accurate results, with the trade-off being computational time.

Appendix B

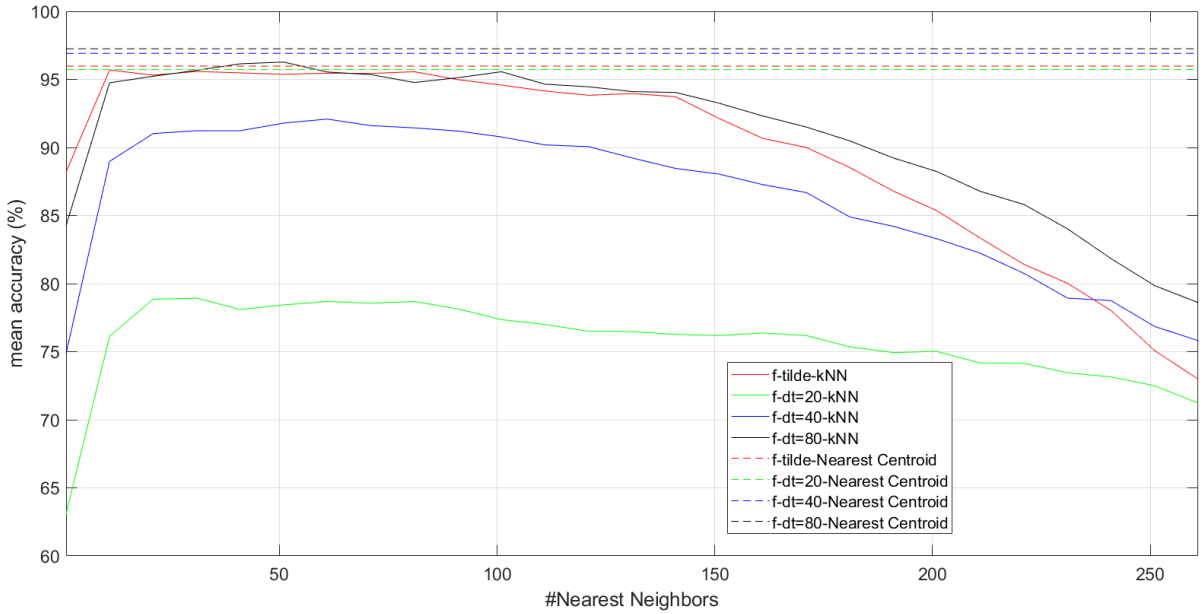


Figure 28 – Comparison of feature vector representations for k-NN at t=320ms, s_P=0.7 using 20 iterations

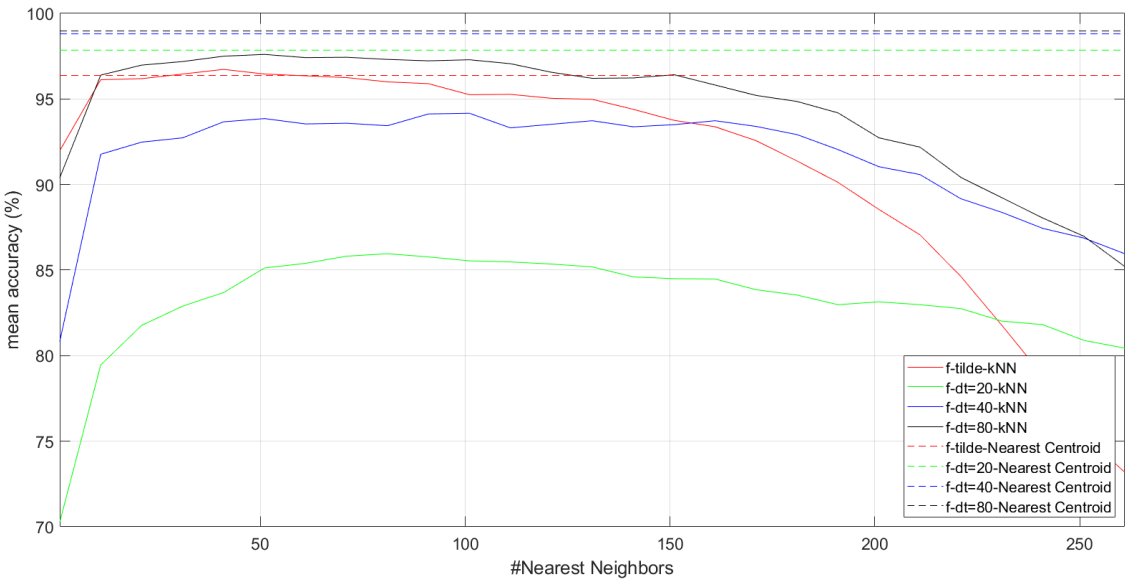


Figure 29 - Comparison of feature vector representations for k-NN at t=560ms, s_P=0.7 using 20 iterations

Representation $\boldsymbol{f}$ produces better results than $\tilde{\boldsymbol{f}}$ in the Nearest Centroid scheme (which outperform the k-NN scheme, see Section V for details). Hence, representation $\boldsymbol{f}$ was used in all classifiers throughout the report.

Appendix C.1 – Code for Position Estimator Training

```
function  [modelParameters] = positionEstimatorTraining(trainingData)
% - trainingData:
%     trainingData(n,k)              (n = trial id,  k = reaching angle)
%     trainingData(n,k).trialId      unique number of the trial
%     trainingData(n,k).spikes(i,t)  (i = neuron id, t = time)
%     trainingData(n,k).handPos(d,t) (d = dimension [1-3], t = time)

r=size(trainingData,1)-1; %Number of Principal Components for PCR

t_train = 320:80:560; %times at which angle classification is updated
classificationParameters = struct;
[F,l,t]=organize_data(trainingData,80,t_train(end)); % create feature vectors for training data
dt=80
for t_ind=1:length(t_train)
    T=t_train(t_ind);
    X=F(t<=T,:);
    M_lda=7;
    M_pca_kNN=170;
    M_pca_SVM=180;
    M_pca_bayes=31;
    % DO PCA using max number of bases (N), the fact that we pre-do it makes the
    % code faster
    [~,mx,Wpca,~]=eigenmodel(X,size(X,2));
    [sb,sw]=make_sb_sw(X,l); %Compute scatter matrices
    M_pca=M_pca_kNN;
    % PCA-LDA for the optimal Mpca for kNN, Mlda=7
    Wopt=make_Wopt(Wpca,M_pca,M_lda,sb,sw);
    W=Wopt'*(X-mx);
    Wmean=zeros([size(W,1) 8]);
    for k=1:8
        Wmean(:,k)=mean(W(:,l==k),2);
    end
    %save model parameters
    classificationParameters(t_ind).Wopt_kNN=Wopt;
    classificationParameters(t_ind).mx_kNN=mx;
    classificationParameters(t_ind).Wmean_kNN=Wmean;

    % PCA-LDA for the optimal Mpca for SVM, Mlda=7
    M_pca=M_pca_SVM;
    Wopt=make_Wopt(Wpca,M_pca,M_lda,sb,sw);
    W=Wopt'*(X-mx);
    Wmean=zeros([size(W,1) 8]);
    for k=1:8
        Wmean(:,k)=mean(W(:,l==k),2);
    end
    %save model parameters
    classificationParameters(t_ind).Wopt_SVM=Wopt;
    classificationParameters(t_ind).mx_SVM=mx;
    classificationParameters(t_ind).Wmean_SVM=Wmean;

    %PCA-LDA for optimal Mpca for bayes, Mlda=7
    M_pca=M_pca_bayes;
    Wopt=make_Wopt(Wpca,M_pca,M_lda,sb,sw);
    W=Wopt'*(X-mx);
    %find the cov-mtrix and mean of the gaussian fitted in data
    A_tot=[];
    C_1_tot=[];
    m_tot=[];
    for k=1:8
        [A,C_1,m]=get_gauss_params(W(:,l==k));
        A_tot=cat(3,A_tot,A);
        C_1_tot=cat(3,C_1_tot,C_1);
        m_tot=cat(3,m_tot,m);
    end
    %save model parameters
    classificationParameters(t_ind).A_bayes=A_tot;
    classificationParameters(t_ind).C_bayes=C_1_tot;
    classificationParameters(t_ind).m_bayes=m_tot;
```

```
    classificationParameters(t_ind).Wopt_bayes=Wopt;
    classificationParameters(t_ind).mx_bayes=mx;
    classificationParameters(t_ind).Wmean_bayes=Wmean;
end

%create feature vectors for training data (dt=20)
T_end = 560;
dt=20;
T = (320:dt:T_end);
[feat,l,t]=organize_data(trainingData,dt,T_end);
%get hand positions from data
[~,~,x,y,~,~]=get_all_handPos(trainingData);
%sample at the values of time we care about
x_resampled = x(:,T,:);
y_resampled = y(:,T,:);

%create a regression model for each angle at each time
%Computing PCR
angle =1:8;
coeffs_gc = struct;
for angle_index =1:length(angle)
    handPos_x_for_regression = x_resampled(:,:,angle_index);
    handPos_y_for_regression = y_resampled(:,:,angle_index);
    for time_index = 1:length(T)
        features_temp = feat(t<=T(time_index),l==angle_index);
        coeffs_gc(time_index, angle_index).mean_hand_pos_x =
mean(handPos_x_for_regression(:,time_index));
        coeffs_gc(time_index, angle_index).mean_hand_pos_y =
mean(handPos_y_for_regression(:,time_index));

        hand_pos_temp_x = handPos_x_for_regression(:,time_index)-coeffs_gc(time_index,
angle_index).mean_hand_pos_x;
        hand_pos_temp_y = handPos_y_for_regression(:,time_index)-coeffs_gc(time_index,
angle_index).mean_hand_pos_y;

        [~,mx,U,~]=eigenmodel(features_temp,r);
        W=U'*(features_temp-mx);
        coeffs_gc(time_index, angle_index).mean_feature=mx;
        bx=U*(W*W')^(-1)*W* hand_pos_temp_x;
        by=U*(W*W')^(-1)*W* hand_pos_temp_y;

        coeffs_gc(time_index, angle_index).values = [bx , by];
    end
end
average_traj = calculate_avg_traj(trainingData);

modelParameters.trajectory = average_traj;
modelParameters.classificationParameters = classificationParameters;
modelParameters.coeffs_gc = coeffs_gc;
modelParameters.r=r;
modelParameters.M_lda=M_lda;
modelParameters.M_pca_kNN=M_pca_kNN;
modelParameters.M_pca_SVM=M_pca_SVM;
modelParameters.M_pca_bayes=M_pca_bayes;
end

% this function calculates the average trajectory
function average_traj = calculate_avg_traj(trainingData)

classes = length(trainingData(1,:));
max_len_spikes = 0; % initialise the longest trial to 0
for c = 1:classes
    for i=1:length(trainingData(:,1))
        % extract the length of the spike trains for each trial
        len_spikes = length(trainingData(i,c).spikes(1,:));
        % check if this length of the spike is the largest
        if len_spikes > max_len_spikes
            max_len_spikes = len_spikes;
        end
    end
end

% make sure all lengths of the trajectories are the same, otherwise we cannot
```

```
% compute the average (number of data points must be the same across
% trajectories of the same class)
for c=1:classes
    for i=1:length(trainingData(:,1))
        for j=length(trainingData(i,c).spikes(1,:))+1:max_len_spikes
            % pad shorter handPos with the last value until
            % max_len_spikes. do not pad with zeros as you would compute the
            % wrong avg, do not cut to shortest trajectory as you would lose
            % information
            trainingData(i,c).handPos = [trainingData(i,c).handPos trainingData(i,c).handPos(:,
end)];
        end
    end
end

% compute the average trajectory
average_traj(classes).handPos = [];
for c=1:classes
    trajectories = zeros(2,max_len_spikes); % two rows, x and y pos
    for i=1:length(trainingData(:,1))
        for j=1:length(trainingData(i,c).handPos(1,:))
            % insert x and y handPos in the trajectory arrays
            trajectories(:,j) = trajectories(:,j) + trainingData(i,c).handPos(1:2,j);
        end
    end
    % compute the average trajectory for each class
    average_traj(c).handPos = trajectories(:,:)/length(trainingData(:,1));
end
end

%this function extracts PCA model parameters
function [N,mx,U,L]=eigenmodel(x,p)
N=size(x,2);
mx=mean(x,2);
A=x-mx;
S=A'*A/N;
[U,L]=eig(S);
p=min(p,size(U,2));
[~,ind]=maxk(diag(L),p);
U=A*U(:,ind);
U=U./sqrt(sum(U.^2));
L=L(ind,ind);
end

%this function makes the within-class/between class scatter matrices
function [sb,sw]=make_sb_sw(X,l)
c=unique(l);
mc=zeros(size(X,1),length(c));
mx=mean(X,2);
for im_id=1:length(c)
    mc(:,im_id)=mean(X(:,l==c(im_id)),2);
end
sb=(mc-mx)*(mc-mx)';
st=(X-mx)*(X-mx)';
sw=st-sb;
end

%create optimal PCA-LDA matrix
function [Wopt]=make_Wopt(Wpca,M_pca,M_lda,sb,sw)
[Wlda,L] = eig((Wpca(:,1:M_pca)'*sw*Wpca(:,1:M_pca))^-1*Wpca(:,1:M_pca)'*sb*Wpca(:,1:M_pca));
[~,ind]=maxk(diag(L),M_lda);
Wopt=Wpca(:,1:M_pca)*Wlda(:,ind);
end

%get scaling factor, covariance and mean for Bayesian
function [A,C_1,m]=get_gauss_params(X)
C=cov(X');
[~,S,~] = svd(C);
temp=S(S~=0);
a=10^(-sum(log10(temp))/length(temp));
C_1=a*(a*C)^-1;
A=-(size(C,1)/2)*log(2*pi/a)-0.5*log(det(a*C));
m=mean(X,2);
```

```
end

%create feature vectors (f)
function [X,l,t]=organize_data(data,dt,T_end)
T=dt:dt:T_end;
X0=zeros([98,size(data,1),size(data,2),length(T)]);

for ind=1:length(T)
    t1=dt*(ind-1)+1;
    t2=dt*ind;
    for k=1:size(data,2)
        for n=1:size(data,1)
            for i=1:98
                X0(i,n,k,ind)=sum(data(n,k).spikes(i,t1:t2))/dt;
            end
        end
    end
end
X1=zeros([size(X0,1)*floor(T(end)/dt) size(X0,2) size(X0,3)]);
t=zeros([1 size(X0,1)*floor(T(end)/dt)]);
for ind=1:floor(T(end)/dt)
    X1(((ind-1)*98+1):((ind-1+1)*98),:,:)=X0(:,:,:,ind);
    t(1,((ind-1)*98+1):((ind-1+1)*98))=T(ind);
end
X=zeros([size(X1,1) size(X1,2)*size(data,2)]);
l=zeros([1 size(X1,2)*size(data,2)]);
for k=1:size(data,2)
    X(:,(k-1)*size(X1,2)+(1:size(X1,2)))=X1(:,:,k);
    l(:,(k-1)*size(X1,2)+(1:size(X1,2)))=k;
end
end

%Extracting hand positions
function[mx,my,x,y,l,in_data]=get_all_handPos(data)
%mx, my = average trajectory
%x,y - row is trials, column is time, 3rd dimension is angle
%l (matrix) - row is length of trial, column is corresponding angle

x_mat=[];
y_mat=[];
for k=1:8
    for n=1:size(data,1)
        x=data(n,k).handPos(1,:);
        l(n,k)=length(x);
        while length(x)>size(x_mat,2) && size(x_mat,1)>0
            x_mat=[x_mat x_mat(:,end)];
        end
        while size(x_mat,2)>length(x) && size(x_mat,1)>0
            x=[x x(end)];
        end
        x_mat=[x_mat;x];
        y=data(n,k).handPos(2,:);
        while length(y)>size(y_mat,2) && size(y_mat,1)>0
            y_mat=[y_mat y_mat(:,end)];
        end
        while size(y_mat,2)>length(y) && size(y_mat,1)>0
            y=[y y(end)];
        end
        y_mat=[y_mat;y];
    end
end
x_mean=[];
y_mean=[];
x=zeros([size(x_mat,1)/8 size(x_mat,2) 8]);
y=zeros([size(y_mat,1)/8 size(y_mat,2) 8]);
for k=1:8
    x(:,:,k)=x_mat(((k-1)*n+1):(k*n),:);
    y(:,:,k)=y_mat(((k-1)*n+1):(k*n),:);
    x_mean=[x_mean;mean(x_mat(((k-1)*n+1):(k*n),:))];
    y_mean=[y_mean;mean(y_mat(((k-1)*n+1):(k*n),:))];
end
mx=zeros([1 size(x_mean,2) size(x_mean,1)]);
my=zeros([1 size(y_mean,2) size(y_mean,1)]);
```

```
for k=1:8
    mx(1,:,k)=x_mean(k,:);
    my(1,:,k)=y_mean(k,:);
end
in_data=zeros(size(x));
for k=1:8
    for n=1:size(in_data,1)
        in_data(n,1:l(n,k),k)=ones(1,l(n,k));
    end
end
end
```

Appendix C.2 – Code for Position Estimator

```
function [x, y, newModelParameters] = positionEstimator(testData, modelParameters)
% - test_data:
%     test_data(m).trialID
%         unique trial ID
%     test_data(m).startHandPos
%         2x1 vector giving the [x y] position of the hand at the start
%         of the trial
%     test_data(m).decodedHandPos
%         [2xN] vector giving the hand position estimated by your
%         algorithm during the previous iterations. In this case, N is
%         the number of times your function has been called previously on
%         the same data sequence.
%     test_data(m).spikes(i,t) (m = trial id, i = neuron id, t = time)
%     in this case, t goes from 1 to the current time in steps of 20

%Saving length of input spike trains
T_end = length(testData.spikes);
% Parameters for SVM
s2=0.07;
c=1;

%Extracting model parameters for classification
classificationParameters=modelParameters.classificationParameters;
if T_end==320 || T_end==400 || T_end==480 || T_end==560
    t_ind=T_end/80-3;
    Xt=organize_data_testing(testData,80,T_end);
    %project features into optimal plane for kNN
    Wopt=classificationParameters(t_ind).Wopt_kNN;
    mx=classificationParameters(t_ind).mx_kNN;
    Wmean=classificationParameters(t_ind).Wmean_kNN;
    Wt=Wopt'*(Xt-mx);
    pred_kNN=do_kNN_fast(2,Wt,Wmean,1:8,1);
    %project features into optimal plane for SVM
    Wopt=classificationParameters(t_ind).Wopt_SVM;
    mx=classificationParameters(t_ind).mx_SVM;
    Wmean=classificationParameters(t_ind).Wmean_SVM;
    Wt=Wopt'*(Xt-mx);
    pred_SVM=do_SVM(Wt,Wmean,s2,c);
    %project features into optimal plane for Bayes
    Wopt=classificationParameters(t_ind).Wopt_bayes;
    mx=classificationParameters(t_ind).mx_bayes;
    Wt=Wopt'*(Xt-mx);
    p=[];
    m_tot=classificationParameters(t_ind).m_bayes;
    A_tot=classificationParameters(t_ind).A_bayes;
    C_1_tot=classificationParameters(t_ind).C_bayes;
    %determine p(features|class) from parameters in training dat
    for k=1:8
        m=m_tot(:,:,k);
        C_1=C_1_tot(:,:,k);
        A=A_tot(:,:,k);
        Y=Wt-m;
        temp1=diag(Y'*C_1*Y);
        p1=exp(A-0.5*temp1);
        p=[p;p1'];
    end
    [~,pred_bayes]=max(p);
    % do majority voting
    [pred_angle,freq]=mode([pred_kNN;pred_SVM;pred_bayes]);
    % if all frequencies are 1, pick SVM
    pred_angle(freq==1)=pred_SVM(freq==1);
else
    %if T_end is not time for update, keep previous label
    pred_angle=modelParameters.test_label;
end


%Regresion (PCR) to estimate hand position
```

```
dt = 20;
T = (320:dt:560);
[features_test]=organize_data_testing(testData,dt,min(T_end,T(end)));
coeffs_gc = modelParameters.coeffs_gc;
coeff_x = coeffs_gc(length(features_test)/98 - 15, pred_angle).values(:,1);
coeff_y = coeffs_gc(length(features_test)/98 - 15, pred_angle).values(:,2);
mx = coeffs_gc(length(features_test)/98 - 15, pred_angle).mean_feature;
mean_hand_pos_x = coeffs_gc(length(features_test)/98 - 15, pred_angle).mean_hand_pos_x;
mean_hand_pos_y = coeffs_gc(length(features_test)/98 - 15, pred_angle).mean_hand_pos_y;

x = (features_test-mx)'*coeff_x+mean_hand_pos_x;
y = (features_test-mx)'*coeff_y+mean_hand_pos_y;
%Note that by construction of the feature vector, if length>560, the result
%will stay at the value it had at 560.
modelParameters.test_label = pred_angle;
newModelParameters = modelParameters;
end

%kNN algorithm
function [labels,err] = do_kNN_fast(ord,test_feat,train_feat_mat,train_lab,NN_vec)
labels=zeros(length(NN_vec),size(test_feat,2));
err=zeros(length(NN_vec),size(test_feat,2));
for n=1:size(test_feat,2)
    test_feat_vec=test_feat(:,n);
    if ord==3
        aa=sqrt(sum(test_feat_vec.^2));%scalar
        ab=sqrt(sum(train_feat_mat.^2));%row vector
        [er1,ind1]=maxk(test_feat_vec'*train_feat_mat./(aa*ab),max(NN_vec));
    else
        [er1,ind1]=mink(sum((abs(test_feat_vec-train_feat_mat)).^ord),max(NN_vec));
    end
    for ind_NN=1:length(NN_vec)
        NN=NN_vec(ind_NN);
        ind=ind1(1:NN);
        er=er1(1:NN);
        train_lab1=train_lab(ind);
        [~,~,temp]=mode(train_lab1);
        er=er(ismember(train_lab1,temp{1}'));
        train_lab1=train_lab1(ismember(train_lab1,temp{1}'));
        [temper,temp]=min(er);
        labels(ind_NN,n)=train_lab1(temp);
        err(ind_NN,n)=temper;
    end
end
end

%SVM implemented using decision tree
function pred = do_SVM(Xt,Xmean,s2,c)
model1 = svmTrain(Xmean', [0 0 1 1 1 1 0 0]', c, @(x1, x2) gaussianKernel(x1, x2, s2));
model2 = svmTrain(Xmean(:,3:6)', [1 1 0 0]', c, @(x1, x2) gaussianKernel(x1, x2, s2));
model3 = svmTrain(Xmean(:,[1 2 7 8])', [1 1 0 0]', c, @(x1, x2) gaussianKernel(x1, x2, s2));
pred=zeros(1,size(Xt,2));
for n=1:size(Xt,2)
    p1=svmPredict(model1,Xt(:,n)');
    if p1==1 %3-6
        p2=svmPredict(model2,Xt(:,n)');
        if p2==1 %3-4
            pred(n)=do_kNN_fast(2,Xt(:,n),Xmean(:,3:4),3:4,1);
        else
            pred(n)=do_kNN_fast(2,Xt(:,n),Xmean(:,5:6),5:6,1);
        end
    else % 1 2 7 8
        p2=svmPredict(model3,Xt(:,n)');
        if p2==1 %1-2
            pred(n)=do_kNN_fast(2,Xt(:,n),Xmean(:,1:2),1:2,1);
        else
            pred(n)=do_kNN_fast(2,Xt(:,n),Xmean(:,7:8),7:8,1);
        end
    end
end
end

%create feature vector (f)
```

```
function [X]=organize_data_testing(data,dt,T_end)
T=dt:dt:T_end;
X=zeros([98*length(T),1]);
for ind=1:length(T)
    t1=dt*(ind-1)+1;
    t2=dt*ind;
    for i=1:98
        X(i+(ind-1)*98,1)=sum(data.spikes(i,t1:t2))/dt;
    end
end
end

%SVM train, from lectures (From Problem Sheet)
function [model] = svmTrain(X, Y, C, kernelFunction, ...
    tol, max_passes)
%SVMTRAIN Trains an SVM classifier using a simplified version of the SMO
%algorithm.
%   [model] = SVMTRAIN(X, Y, C, kernelFunction, tol, max_passes) trains an
%   SVM classifier and returns trained model. X is the matrix of training
%   examples.  Each row is a training example, and the jth column holds the
%   jth feature.  Y is a column matrix containing 1 for positive examples
%   and 0 for negative examples.  C is the standard SVM regularization
%   parameter.  tol is a tolerance value used for determining equality of
%   floating point numbers. max_passes controls the number of iterations
%   over the dataset (without changes to alpha) before the algorithm quits.
%
% Note: This is a simplified version of the SMO algorithm for training
%       SVMs. In practice, if you want to train an SVM classifier, we
%       recommend using an optimized package such as:
%
%           LIBSVM   (http://www.csie.ntu.edu.tw/~cjlin/libsvm/)
%           SVMLight (http://svmlight.joachims.org/)
%
%

if ~exist('tol', 'var') || isempty(tol)
    tol = 1e-3;
end

if ~exist('max_passes', 'var') || isempty(max_passes)
    max_passes = 5;
end

% Data parameters
m = size(X, 1);
n = size(X, 2);

% Map 0 to -1
Y(Y==0) = -1;

% Variables
alphas = zeros(m, 1);
b = 0;
E = zeros(m, 1);
passes = 0;
eta = 0;
L = 0;
H = 0;

% Pre-compute the Kernel Matrix since our dataset is small
% (in practice, optimized SVM packages that handle large datasets
%  gracefully will _not_ do this)
%
% We have implemented optimized vectorized version of the Kernels here so
% that the svm training will run faster.
if strcmp(func2str(kernelFunction), 'linearKernel')
    % Vectorized computation for the Linear Kernel
    % This is equivalent to computing the kernel on every pair of examples
    K = X*X';
elseif contains(func2str(kernelFunction), 'gaussianKernel')
    % Vectorized RBF Kernel
    % This is equivalent to computing the kernel on every pair of examples
    X2 = sum(X.^2, 2);
```

```
    K = bsxfun(@plus, X2, bsxfun(@plus, X2', - 2 * (X * X')));
    K = kernelFunction(1, 0) .^ K;
else
    % Pre-compute the Kernel Matrix
    % The following can be slow due to the lack of vectorization
    K = zeros(m);
    for i = 1:m
        for j = i:m
            K(i,j) = kernelFunction(X(i,:)', X(j,:)');
            K(j,i) = K(i,j); %the matrix is symmetric
        end
    end
end

% Train
% fprintf('\nTraining ...');
dots = 12;
while passes < max_passes

    num_changed_alphas = 0;
    for i = 1:m

        % Calculate Ei = f(x(i)) - y(i) using (2).
        % E(i) = b + sum (X(i, :) * (repmat(alphas.*Y,1,n).*X)') - Y(i);
        E(i) = b + sum (alphas.*Y.*K(:,i)) - Y(i);

        if ((Y(i)*E(i) < -tol && alphas(i) < C) || (Y(i)*E(i) > tol && alphas(i) > 0))

            % In practice, there are many heuristics one can use to select
            % the i and j. In this simplified code, we select them randomly.
            j = ceil(m * rand());
            while j == i  % Make sure i \neq j
                j = ceil(m * rand());
            end

            % Calculate Ej = f(x(j)) - y(j) using (2).
            E(j) = b + sum (alphas.*Y.*K(:,j)) - Y(j);

            % Save old alphas
            alpha_i_old = alphas(i);
            alpha_j_old = alphas(j);

            % Compute L and H by (10) or (11).
            if (Y(i) == Y(j))
                L = max(0, alphas(j) + alphas(i) - C);
                H = min(C, alphas(j) + alphas(i));
            else
                L = max(0, alphas(j) - alphas(i));
                H = min(C, C + alphas(j) - alphas(i));
            end

            if (L == H)
                % continue to next i.
                continue;
            end

            % Compute eta by (14).
            eta = 2 * K(i,j) - K(i,i) - K(j,j);
            if (eta >= 0)
                % continue to next i.
                continue;
            end

            % Compute and clip new value for alpha j using (12) and (15).
            alphas(j) = alphas(j) - (Y(j) * (E(i) - E(j))) / eta;

            % Clip
            alphas(j) = min (H, alphas(j));
            alphas(j) = max (L, alphas(j));

            % Check if change in alpha is significant
            if (abs(alphas(j) - alpha_j_old) < tol)
                % continue to next i.
```

```
                % replace anyway
                alphas(j) = alpha_j_old;
                continue;
            end

            % Determine value for alpha i using (16).
            alphas(i) = alphas(i) + Y(i)*Y(j)*(alpha_j_old - alphas(j));

            % Compute b1 and b2 using (17) and (18) respectively.
            b1 = b - E(i) ...
                - Y(i) * (alphas(i) - alpha_i_old) *  K(i,j)' ...
                - Y(j) * (alphas(j) - alpha_j_old) *  K(i,j)';
            b2 = b - E(j) ...
                - Y(i) * (alphas(i) - alpha_i_old) *  K(i,j)' ...
                - Y(j) * (alphas(j) - alpha_j_old) *  K(j,j)';

            % Compute b by (19).
            if (0 < alphas(i) && alphas(i) < C)
                b = b1;
            elseif (0 < alphas(j) && alphas(j) < C)
                b = b2;
            else
                b = (b1+b2)/2;
            end

            num_changed_alphas = num_changed_alphas + 1;

        end

    end

    if (num_changed_alphas == 0)
        passes = passes + 1;
    else
        passes = 0;
    end

    %     fprintf('.');
    dots = dots + 1;
    if dots > 78
        dots = 0;
        fprintf('\n');
    end
    if exist('OCTAVE_VERSION')
        fflush(stdout);
    end
end
% fprintf(' Done! \n\n');

% Save the model
idx = alphas > 0;
model.X= X(idx,:);
model.y= Y(idx);
model.kernelFunction = kernelFunction;
model.b= b;
model.alphas= alphas(idx);
model.w = ((alphas.*Y)'*X)';
end

%gaussian kernel, from lectures (From Problem Sheet)
function sim = gaussianKernel(x1, x2, sigma)
%RBFKERNEL returns a radial basis function kernel between x1 and x2
%   sim = gaussianKernel(x1, x2) returns a gaussian kernel between x1 and x2
%   and returns the value in sim

% Ensure that x1 and x2 are column vectors
x1 = x1(:); x2 = x2(:);

sim = exp(-(norm(x1 - x2) ^ 2) / (2 * (sigma ^ 2)));

end

%SVM predict, from lectures (From Problem Sheet)
```

```
function pred = svmPredict(model, X)
%SVMPREDICT returns a vector of predictions using a trained SVM model
%(svmTrain).
%   pred = SVMPREDICT(model, X) returns a vector of predictions using a
%   trained SVM model (svmTrain). X is a mxn matrix where there each
%   example is a row. model is a svm model returned from svmTrain.
%   predictions pred is a m x 1 column of predictions of {0, 1} values.
%

% Check if we are getting a column vector, if so, then assume that we only
% need to do prediction for a single example
if (size(X, 2) == 1)
    % Examples should be in rows
    X = X';
end

% Dataset
m = size(X, 1);
p = zeros(m, 1);
pred = zeros(m, 1);

if strcmp(func2str(model.kernelFunction), 'linearKernel')
    % We can use the weights and bias directly if working with the
    % linear kernel
    p = X * model.w + model.b;
elseif contains(func2str(model.kernelFunction), 'gaussianKernel')
    % Vectorized RBF Kernel
    % This is equivalent to computing the kernel on every pair of examples
    X1 = sum(X.^2, 2);
    X2 = sum(model.X.^2, 2)';
    K = bsxfun(@plus, X1, bsxfun(@plus, X2, - 2 * X * model.X'));
    K = model.kernelFunction(1, 0) .^ K;
    K = bsxfun(@times, model.y', K);
    K = bsxfun(@times, model.alphas', K);
    p = sum(K, 2);
else
    % Other Non-linear kernel
    for i = 1:m
        prediction = 0;
        for j = 1:size(model.X, 1)
            prediction = prediction + ...
                model.alphas(j) * model.y(j) * ...
                model.kernelFunction(X(i,:)', model.X(j,:)');
        end
        p(i) = prediction + model.b;
    end
end

% Convert predictions into 0 / 1
pred(p >= 0) =  1;
pred(p <  0) =  0;

end
```