Chapter 7

Experiment Result Analysis

In this chapter we will analyze the results of the different experiments executed using the created Framework. As mentioned in chapter 5, the experiments fall on 4 different categories:

- Benchmark Comparison
- Preprocess Component Analysis
- Learning Algorithm Analysis
- Multi-classification on HLA DataSet

7.1. Benchmark comparison

As a validation method for the implementation of the framework, we compared the results we obtained on both the handwritten digit recognition dataset and the secondary structure in proteins dataset against benchmark measurements. For the first dataset, results using k-means clustering [3] are shown in table 7.1. The results provided are based on different values for k, and represent the accuracy of the complete multi-class classifier. For our multi-classifier, after running adaboost* 1319 iterations, we had an

k	*Accuracy (%)*
1	98
2	97.38
3	97.83
4	97.61
5	97.89
6	97.77
7	97.66
8	97.66
9	97.72
10	97.55
11	97.89

Table 7.1: k nearest neighbors results. [3]

accuracy of 97.92 %, which is only improved with k-nearest neighbors when k=1, where the accuracy was 98 %.

For the case of the secondary structure in proteins prediction, an available result [7] indicated a generalization error of 29,48 % using Adaboost with 1000 iterations. Using 1319 iterations of Adaboost* with precision of 0.05, we obtained a generalization error of 27,97 %, with a final margin of −0,0251. Thus, we improved the generalization error by 2,51 %.

7.2. Preprocess Component Analysis

A special interest in our research was comparing different preprocessing techniques and evaluating their effect on the final performance of the agent system. For this purpose, 2 experiments were performed using the handwritten digits dataset (Figure 7.1). The first one used hard partition, making only one feature available to each agent, being that agent the only one with access to it. The second experiment used soft partition, where each agent had access to 2 consecutive features, which could also be accessed by

another agent.

7.2.1. Hard Partition Experiment

In our hard partition experiment, we had 3823 samples in our training set and 896 different agents to choose from, each with access to a binary feature. Each sample had originally 64 features with values 0 ... 16. For the experiment, each feature was expanded into 17 features by converting the feature value into a 17-digit string, which was 0-valued excepting for one position which corresponds to the original value of the feature. For instance, a feature with value 7 would become the set of features $\{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$.

Adaboost* was run with the set of agents for 100 iterations, building a binary classifier for each of the 10 digits. The generalization error for each of the binary classifiers is shown in table 7.2. From this data, we observe that for certain digits, as 0,6 and 7, the generalization error was low compared to other digits such as 8,9 which presented more problems for classification. The results of each binary classifier reflect on the final multiclass classifier, as it is shown in table 7.3, which shows the confusion matrix for the multi-classifier. The multi-classifier had a generalization error of 11,07 % on 1797 unseen test samples.

From the data in the confusion matrix in table 7.3 we may make the following observations. Digits 0,5,6 had good classification results, as over 95 % of the true positives were classified as positives. On the other hand, 1,8,9 had the worst result, being less than 85 % of true positives classified as such. Looking now at the classifiers, classifiers for 0,5 and 6 were good as over 95 % of their true positive predictions were true positives. On the other hand, classifiers for 1,8 and 9 were not good as less than 85 % of the predictions were true positives. This matrix shows how the quality of each single binary

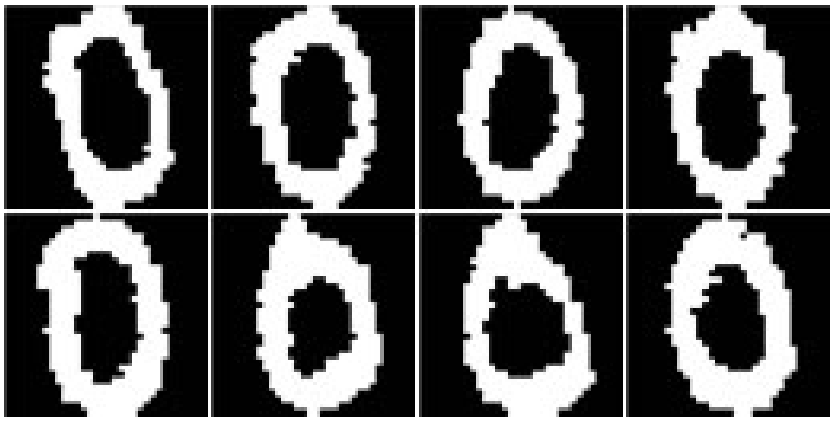

Figure 7.1: Sample of handwritten '0'digits

Digit	Generalization Error	Training Error
0	0.61 %	0.71 %
1	3.78 %	2.20 %
2	2.61 %	1.96 %
3	3.39 %	2.83 %
4	2.61 %	2.64 %
5	1.66 %	2.54 %
6	1.78 %	1.12 %
7	2.50 %	1.15 %
8	6.06 %	5.34 %
9	5.11 %	4.53 %

Table 7.2: Generalization and training error per binary classifier for hard partition test

	0	1	2	3	4	5	6	7	8	9
0	175	0	0	0	1	0	0	0	2	0
1	0	136	16	5	4	3	0	0	5	13
2	1	2	156	9	2	1	0	1	5	0
3	2	2	3	156	0	2	1	4	3	10
4	0	2	0	0	174	0	1	3	0	1
5	0	1	0	2	0	172	3	1	2	1
6	0	4	0	0	3	0	173	0	1	0
7	0	0	0	0	3	0	0	162	5	9
8	0	14	1	1	3	0	0	0	140	15
9	3	2	0	8	4	1	0	3	3	156

Table 7.3: Confusion matrix for hard partition test

classifier affects the complete ensemble. 8 and 9 had the worst binary classifiers and hence the multi-classifier has the most trouble with these digits. 0 had the best binary classifier and that reflects on the good performance of the multi-classifier on this digit. However, this is not always true. In the case of digit 1, the binary classifier seems to be better than those of other digits, however, the performance on this digit is poor. This is a consequence of the 8 and 9 classifiers which produce the highest valued prediction in 5 and 13 times respectively in the presence of the digit 1. Interestingly enough, the classifier for 1 provides the highest prediction value in the presence of digit 8 14 times, making it clear that our multi-classifier often mistakes these 2 digits with one another.

7.2.2. Soft Partition Experiment

For the case of soft partition, a second processing was performed on the data. In this case, given the binary representation of 2 features, the linear combination is generated and each pair is assigned to an agent. This way, each agent has access to information of two different features. For instance, consider two consecutive features in the original dataset x_i and x_{i+1}. For each possible value pair (x_i, x_{i+1}), $x_i \in [0, 16]$, $x_{i+1} \in [0, 16]$ there will be an agent with access to the pair. This will be the case too for the pair x_{i+1} and x_{i+2}, such that there will be at least 2 agents with access to each value of x_{i+1}. Originally, 18496 agents were generated, and a further simplification was performed to reduce the number of features. Instead of having 17 possible values for a feature, 5 value ranges were defined so that each feature will have 5 possible values. After this procedure, and after applying the soft partition method mentioned beforehand, a total of 1600 agents were generated.

With this new set of agents, Adaboost* was again executed. The results for individual binary classifiers are shown in table 7.4. The multi-classifier had a generalization

Digit	Generalization Error	Training Error
0	0.95 %	0.86 %
1	4.67 %	3.03 %
2	2.39 %	1.60 %
3	3.73 %	2.64 %
4	1.95 %	2.85 %
5	1.34 %	1.41 %
6	1.50 %	0.84 %
7	2.34 %	1.15 %
8	5.95 %	4.58 %
9	4.73 %	3.30 %

Table 7.4: Generalization and training error per binary classifier for soft partition test

	0	1	2	3	4	5	6	7	8	9
	0	1	2	3	4	5	6	7	8	9
0	176	0	0	0	2	0	0	0	0	0
1	0	146	9	4	5	1	0	0	4	13
2	0	0	163	3	2	0	0	1	7	1
3	1	3	0	155	0	1	0	1	12	10
4	0	10	2	0	165	0	0	2	1	1
5	2	0	0	2	1	169	3	0	0	5
6	2	2	0	0	1	2	172	0	2	0
7	0	1	0	0	5	3	0	155	9	6
8	0	8	0	1	2	1	1	1	145	15
9	1	1	1	6	5	3	0	3	8	152

Table 7.5: Confusion matrix for soft partition test

error of 10,96 %, and its performance may be evaluated from the confusion matrix in table 7.5.

From the results we may make the following observations. Binary classifiers for 0,5 and 6 presented again the best results, while 1,8 and 9 had the highest generalization error. In the confusion matrix, the classifier again performed best for 0,5,6 and was weak for 1,8,9.

7.2.3. Hard - Soft Partition Result Comparison

The focus of this experiment was to compare the performance of soft and hard partition on the handwritten digits dataset. In general terms, soft partition performed better, achieving a generalization error of 10,96 %, while hard partition achieved 11,07 %. However, this improvement is small. If we focus our analysis on comparing binary classifiers and analyzing in which cases having access to more than one feature is more useful, we will gather more interesting information. We will focus on the case of two digits: 1 and 5.

In the case of the digit 1, table 7.6 shows the comparative data of soft and hard partition. Using hard partition the results are far superior. Both generalization and training errors are lower; the number of true positives is higher and the number of false positives lower. Given these result, arises the question: Why is the performance of hard partition better that soft partition? Figure 7.2 might give us some insight on the answer. In the figure we see several handwritten '1'digits. It is clear that the ways this digit may be written are diverse, making it hard for our classifier, which focuses on the information of pixels. Our results indicate that having access to two consecutive pixels (features) is worse than only having access to one. This reflects that pixels by themselves are more effective for recognizing a digit 1 than pairs. The figures show that the location of white/dark edges varies considerably from one digit to another, which may explain why pairs are not being useful. Figure 7.3 provides more information. It shows the progression of γ through the boosting process. It is important to keep in mind that $\epsilon_t = \frac{1}{2} - \frac{1}{2}\gamma_t$, where ϵ_t is the minimum error an agent has on the reweighted dataset. As gamma is reduced, the error increases and hence the contribution of agents is less. In figure 7.3 we can see how the value of γ falls dramatically for soft partition against hard partition. This means that the contribution of agents, even when it was higher in

Per Digit Comparison		
	Hard Partition	Soft Partition
Generalization Error	3.78 %	4.67 %
Training Error	2.20 %	3.03 %
System Correct Positives	146	136
System False Positives	25	27

Table 7.6: Comparative results of the binary classifier for digit 1

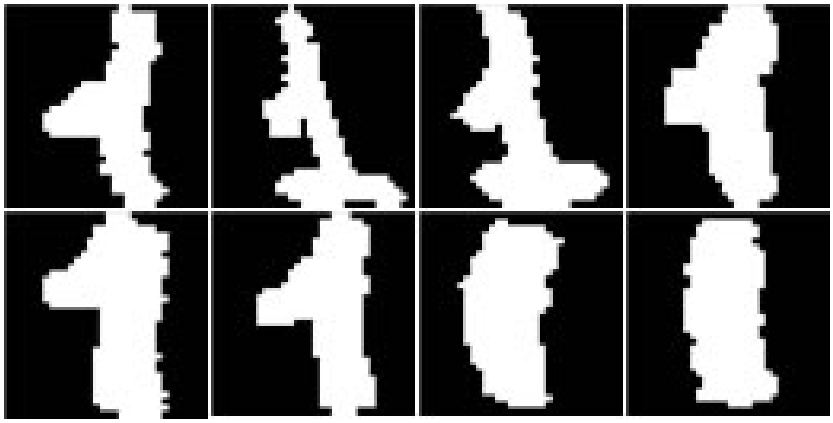

Figure 7.2: Sample of handwritten '1'digits

the first iterations, is much less in soft partition than in hard partition as the iterations continue. We may attribute this to the fact that, as agents share information, newly chosen agents don't have that much to contribute, as their prediction is not that different from the one of previously chosen agents. On the other hand, with hard partition, agents do differ considerably and can provide more information as the iterations advance.

If we look at the case of the digit 5, we observe different results. 5 was for both experiments one of the digits where the best performance was achieved. The comparison data between soft and hard partition is shown in table 7.7. In this case soft partition is superior, as both generalization and training errors are lower, correct positives are higher and incorrect positives are lower. These results hint us that in the case of digit '5ágents which considered pairs were able to provide us more useful information than those which only looked at a single agent. Looking at figure 7.4, we see that the different

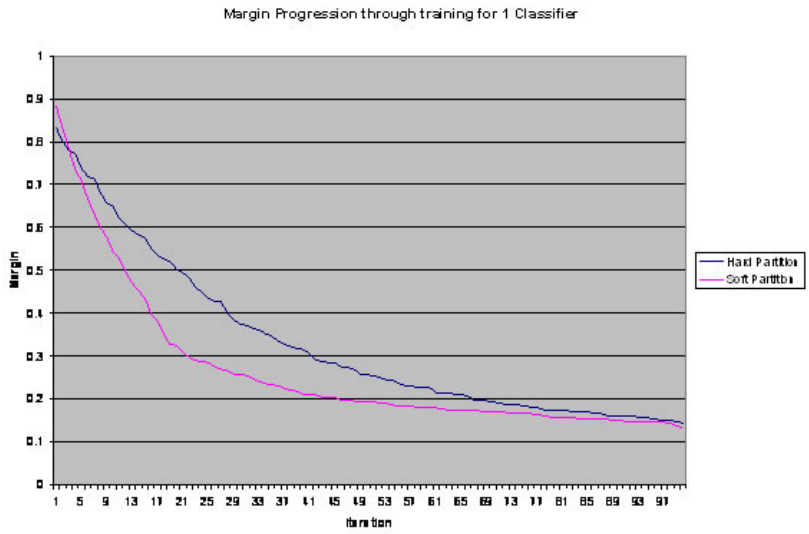

Figure 7.3: Gamma progression through boosting algorithm for binary classifier for digit '1'. Results for hard and soft partitioning are graphed.

Per Digit Comparison		
	Hard Partition	Soft Partition
Generalization Error	1.66 %	1.34 %
Training Error	2.54 %	1.41 %
System Correct Positives	169	172
System False Positives	11	7

Table 7.7: Comparative results of the binary classifier for digit 5

'5'samples are more similar than in the case of the digit 1, and that there are common areas where the figure's edges are found, which are useful for paired features. Figure 7.5 provides more grounds to our analysis as in this case γ is higher for soft partition throughout the 100 iterations, which means that agents which have access to a feature pair are more useful than those with access to a single feature.

From this experiment, we conclude that there is a close relationship between the problem at hand and the convenience of hard and soft partition. However, we could observe that cases where certain features are present for different samples and those features can be represented by looking at more than one feature (as in the case of

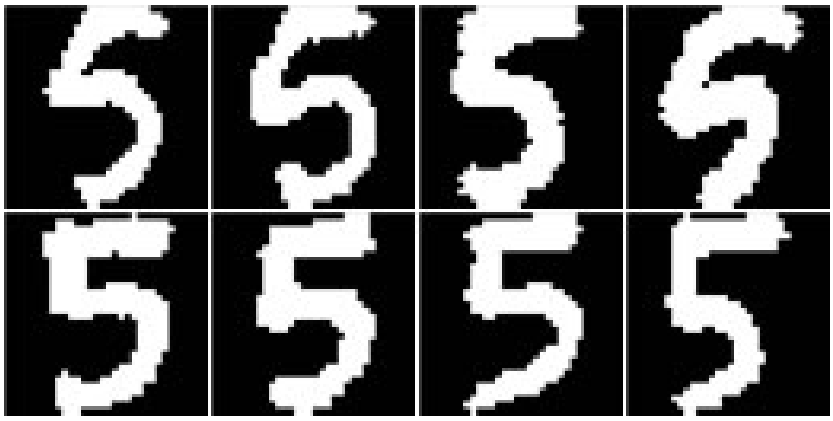

Figure 7.4: Sample of handwritten '5'digits

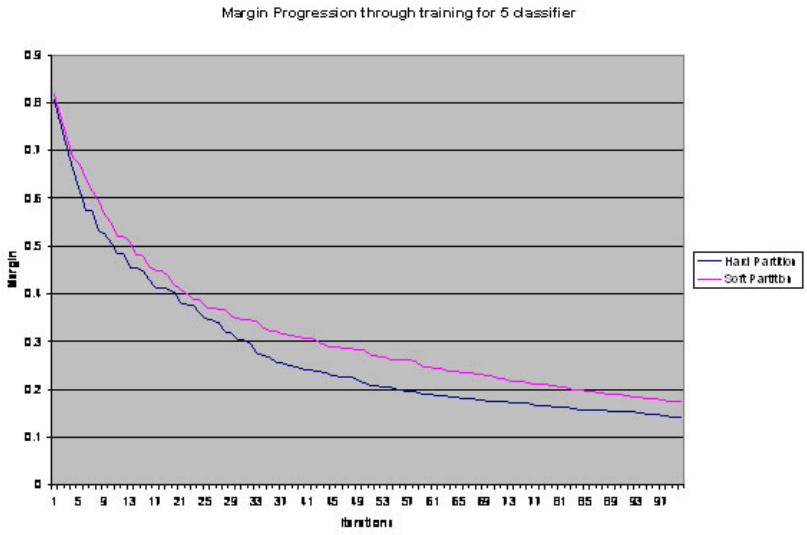

Figure 7.5: Gamma progression through boosting algorithm for binary classifier for digit '5'. Results for hard and soft partitioning graphed.

figure edges) seem to be more appropriate for soft partition. When using boosting, the progression of γ is an useful tool when deciding which approach is better.

7.3. Learning Algorithm Analysis

It was relevant to analyze the behavior of different available ensemble algorithms and compare them against each other. The results of this analysis provided us grounds in choosing an ensemble technique to use for the HLA dataset. This analysis consisted on 2 different experiments. In the first one, we compared the Adaboost, Marginal Adaboost and Adaboost* algorithms on the handwritten digits dataset. In the second one, we analyzed Bagging on the Boston house cost dataset.

7.3.1. Boosting Algorithm Analysis

Marginal Adaboost was initially run with our dataset for the recognition of the 10 different digits. The total number of iterations, number of iterations in the last $Adaboost_\gamma$ call, final margin, final edge and final generalization error for each case were collected. The results with precision $\epsilon = 0{,}5$ are shown in table 7.8. Figure 7.6 compares the margin versus de generalization error.

Interesting to notice from our results is that in average they are similar to the best results of k nearest neighbors. Also, it is interesting to observe how in the case of '0'the number of iterations was significantly higher than the number of iterations with the other digits. We may attribute this to the fact that $\varrho*$ may actually be higher in this case, being 0 a number significantly different from other numbers. Note that we reached a margin of 0.0017 in this case, which means that $\varrho*$ is at most 0.0517, since we used a precision of 0.05 .

The obtained generalization error also tells us about how easy a digit is to identify.

digit	*Total It.*	*Last Adaboost$_\gamma$ It.*	$\hat{\varrho}$	$\hat{\gamma}$	*G. Error (%)*
0	6656	47	0.0017	0.2383	0.17
1	392	82	-0.0490	0.1009	1.95
2	138	55	-0.0471	0.1381	1.45
3	639	95	-0.0449	0.0999	3.23
4	273	79	-0.0395	0.1155	1.5
5	390	99	-0.0474	0.1322	1.22
6	170	61	-0.0479	0.1726	0.89
7	146	56	-0.0367	0.1633	1.73
8	801	114	-0.0492	0.0865	3.78
9	1923	166	-0.0479	0.0548	4.12
mean	1152.8	85.4	-0.0408	0.1302	2
sd	1005.8321	35.7497	0.0155	0.0519	1.29

Table 7.8: Results of Marginal Adaboost on dataset

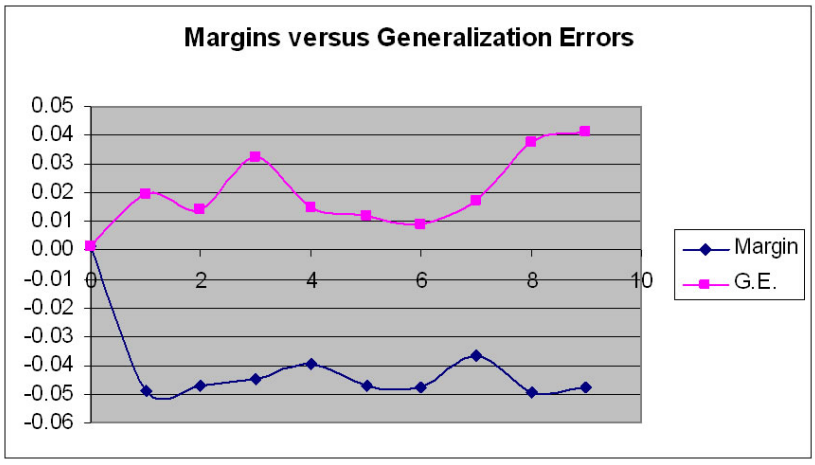

Figure 7.6: Graph plot of the generalization error and margin for the binary classifier for each digit.

digit s	*its.* s	Marginal *Margin*	*G.Error (%)*	Adaboost *Margin*	*G.Error(%)*	Adaboost* *Margin*	*G.Error(%)*
0	47	0.0017	0.17	-0.4605	1.22	0.2894	0.95
1	82	-0.0490	1.95	-0.5546	4.34	0.1306	4.06
2	55	-0.0471	1.45	-0.3579	3.67	0.2402	3.06
3	95	-0.0449	3.23	-0.3245	3.90	0.0967	3.51
4	79	-0.0395	1.5	-0.3467	2.84	0.1307	3.01
5	99	-0.0474	1.22	-0.4367	1.89	0.0904	1.56
6	61	-0.0479	0.89	-0.5208	3.01	0.2319	2.23
7	56	-0.0367	1.73	-0.5251	3.62	0.2186	3.67
8	114	-0.0492	3.78	-0.2987	5.84	0.0621	5.40
9	166	-0.0479	4.12	-0.3369	5.23	0.0168	4.56
mean	85.4	-0.0408	2	-0.4162	3.57	0.15074	3.2
sd	35.7497	0.0155	1.29	0.09491	1.41	0.0892	1.35

Table 7.9: Comparison of results. Marginal Adaboost complete run, Adaboost and Adaboost* after the number of iterations in the final $adaboost_\varrho$ request.

0 and 6 got the best results, which we may understand as they have a very distinctive shape different from the other digits. 3,8 and 9 had greater generalization error, which we may attribute to their similarities to other of the digits.

After the results of marginal adaboost were obtained, Adaboost and Adaboost* were run, measuring the margin ϱ and the generalization error at three different moments: after the total iterations of the last call of $Adaboost_\varrho$ in marginal adaboost, after the overall number of iterations of $MarginalAdaboost$ and after 1319 iterations. We will now compare results after each one of the three moments.

In the first test (table 7.9), Marginal Adaboost obtained the best results. We can attribute this to the fact that in the last iteration, the estimated ϱ used in the $Adaboost_\varrho$ request is close to the optimal ϱ. We can also see that the margins are quite different in the 3 methods. We believe $Adaboost_\varrho$ has the most correct margin, and the margin for regular adaboost is far below, while the margin for Adaboost* is above (remember Adaboost*, different from marginal adaboost, calculates the margins in a decreasing

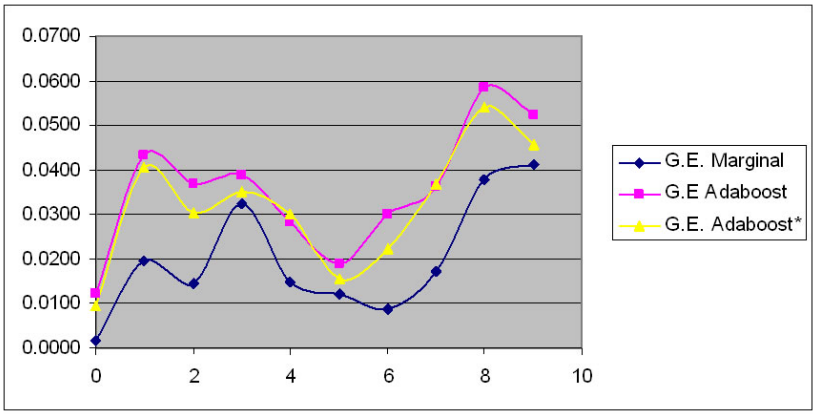

Figure 7.7: Graphical comparison of generalization error for adaboost, marginal adaboost and adaboost* for the number of iterations in the final marginal adaboost's $adaboost_\varrho$ call.

fashion). A graphical comparison of the generalization error for the three algorithms can be seen in figure 7.7.

Finally, notice how the generalization error of Adaboost* is already better than the generalization error of Adaboost, even after a small number of iterations.

Now we analyze the results after the total number of iterations of Marginal Adaboost. These are all the boosting iterations performed by the algorithm, not only the iterations in the last $adaboost_\rho$ request (Table 7.10)

Again, Marginal Adaboost shows better results, though Adaboost* is able to beat Marginal Adaboost in some cases. Interestingly enough, these cases are the ones where the margin is brought down the most (3,8). Again, regular Adaboost has the worst generalization error. The graphical comparison of the generalization errors can be seen in figure 7.8.

Although the results with Marginal Adaboost were better, we still have not considered the total number of iterations required to minimize the margin for Adaboost* (1319 in this case). In some of the cases, the total number of iterations is less than the required number of iterations for Adaboost, which shows how in some cases Marginal

digit s	*its.* s	Marginal *Margin*	*G.Error (%)*	Adaboost *Margin*	*G.Error(%)*	Adaboost* *Margin*	*G.Error(%)*
0	6656	0.0017	0.17	0.0150	0.22	0.0537	0.28
1	392	-0.0490	1.95	-0.2708	2.95	0.0183	2.06
2	138	-0.0471	1.45	-0.2590	2.62	0.1039	2.11
3	639	-0.0449	3.23	-0.1428	3.39	0.0086	3.01
4	273	-0.0395	1.5	-0.2042	2.50	0.0361	1.78
5	390	-0.0474	1.22	-0.2242	1.34	0.0231	1.28
6	170	-0.0479	0.89	-0.3512	1.67	0.0923	1.17
7	146	-0.0367	1.73	-0.3795	2.78	0.0838	2.11
8	801	-0.0492	3.78	-0.1086	4.28	-0.0038	3.51
9	1923	-0.0479	4.12	-0.0894	4.12	0.0168	3.51
mean	1152.8	-0.0408	2	-0.20147	2.587	0.0397	2.08
sd	2005.8321	0.0155	1.29	0.1221	1.25	0.0422	1.04

Table 7.10: Comparison of results. Marginal Adaboost complete run, Adaboost and Adaboost* after total number of iterations of Marginal Adaboost.

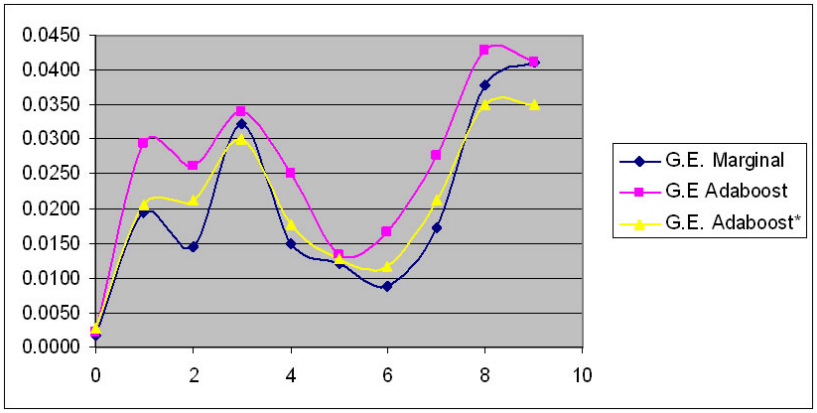

Figure 7.8: Graphical comparison of generalization error for adaboost, marginal adaboost and adaboost* for the total number of iterations in marginal adaboost.

digit s	*its.* s	Marginal *Margin*	*G.Error (%)*	Adaboost *Margin*	*G.Error(%)*	Adaboost* *Margin*	*G.Error(%)*
0	1319	0.0017	0.17	-0.0729	0.33	0.0602	0.17
1	1319	-0.0490	1.95	-0.1558	2.50	0.0070	1.89
2	1319	-0.0471	1.45	-0.0907	1.78	0.0184	1.22
3	1319	-0.0449	3.23	-0.1145	3.39	-0.0010	2.84
4	1319	-0.0395	1.5	-0.0968	1.84	0.0091	1.34
5	1319	-0.0474	1.22	-0.1287	1.34	0.0068	1.11
6	1319	-0.0479	0.89	-0.1344	0.83	0.0258	0.67
7	1319	-0.0367	1.73	-0.1129	1.95	0.0245	1.50
8	1319	-0.0492	3.78	-0.0957	4.23	-0.0052	3.34
9	1319	-0.0479	4.12	-0.1365	4.28	-0.0176	3.45
mean	1319	-0.0408	2	-0.1139	2.24	0.0128	1.753
sd	0	0.0155	1.29	0.0253	1.35	0.0214	1.16

Table 7.11: Comparison of results. Marginal Adaboost complete run, Adaboost and Adaboost* after 1319 iterations.

Adaboost might turn out better results in some cases where the margin is not too big and we are looking for a small number of iterations.

Our final set of results shows the results of Adaboost* after the required number of iterations to reach $\varrho*$ with precision ϵ. We compare the results of Marginal Adaboost against Adaboost* and Adaboost after 1319 iterations, which is the number of iterations indicated to be necessary for Adaboost* to find a margin with distance at most ϵ from $\varrho*$ ($\lceil \frac{2\log(N)}{v^2} + 1 \rceil$). (Table 7.11)

In the final results, we may clearly see how Adaboost* obtains the best generalization error in all of the cases. Notice how the final margins of Adaboost* are all small, reflecting the fact that the margins in our data are small $\varrho*$.

In this final comparison, it is also interesting to analyze the margins of the regular adaboost compared to the margins of Adaboost*. It has been proven that asymptotically Adaboost achieves a margin of at least $\varrho*/2$. We can clearly see from the three runs, how by increasing the number of iterations, the margin reached by Adaboost also increases. Specifically, on table 4 in the first run which required 6500 iterations, we can see how

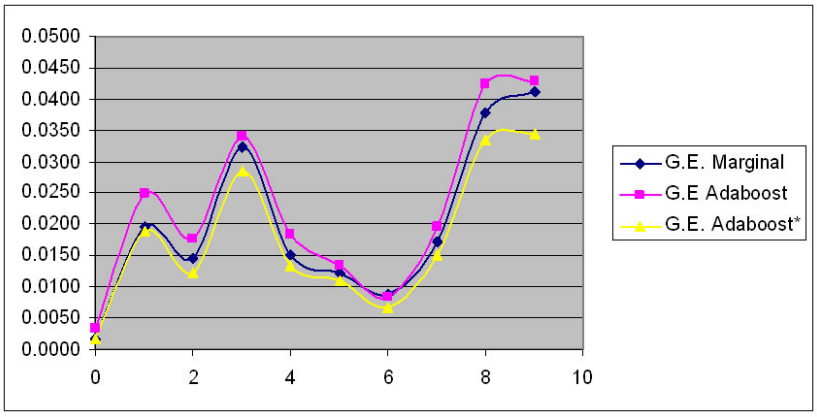

Figure 7.9: Graphical comparison of generalization error for adaboost, marginal adaboost and adaboost*. Regular Marginal adaboost run, adaboost* and adaboost run for 1319 iterations.

the margin from Adaboost is close to the margin from Marginal Adaboost. A graphical comparison of generalization errors can be seen in figure 7.9.

Another interesting observation we may make has to do with the case of detecting the digit 9 with Adaboost star. In table 4, after 1923 iterations, the generalization error was 3,51 %, however after only 1319 iterations the generalization error was 3,45 %; This is a sign of the overfitting risk when using Adaboost*. Using more iterations than the required can translate into overfitting. This can also be seen regarding the classification of the number 1, where with 6656 iterations of Adaboost* the generalization error is 0,28 %, but after 1319 iterations was 0,17 %.

Marginal Adaboost and Adaboost* are clear improvements on the general Adaboost Algorithm. Marginal Adaboost is able to reach a considerably low generalization error using just a few iterations in its final $Adaboost_{\varrho}$ call, which showcases how powerful having a good estimate of the margin can be. Marginal Adaboost has the clear disadvantage that in cases when the margin is too high it might take too long to find it. Adaboost* overcomes those difficulties as in one Adaboost run it maximizes the margin, returning in the end the best results. In the end, both methods return better results

Error Ratio Limit/Total Agents	10	20	30
1	12.9700817569586	12.9230717136202	12.9525885255181
0.5	6.58880066004197	6.59636749198932	6.50571841801899
0.49	6.44720404379967	6.4332411895325	6.44546495472569

Table 7.12: Average Error for Agent System. The Average Error is the average absolute difference among the real value y and the prediction for all data

than the regular Adaboost.

It is important though, to be careful with problems like overfitting, which were present in our Adaboost* results.

7.4. Bagging algorithm analysis

For the implementation of bagging, a Neural Network was trained for each one of the agents in the complete ensemble. The interest in the analysis was to compare the performance of one agent against the ensemble, and with it to bring empirical evidence to the bagging conception which indicates that in average, a set of Agents will perform better than a single Agent on certain datasets.

For the experiment, bagging was initially run with 10,20 and 50 agents, and with an error ratio of 1, 0.5 and 0.49. The average error of the resulting systems is shown in table 7.12.

The results in table 7.12 do not provide us with any specific insight on the performance of the algorithm when the number of agents increase. This is a consequence of only needing a small set of agents for all the representative samples to be covered. Hence, increasing the number of agents does not provide any clear improvement on the average error.

A second experiment was then performed, this time focusing on the performance of single agents against the ensemble system. In the results, the average error of each of the

Agent Id	Average Error
0	6.470720424
1	6.383580591
2	6.317212682
3	6.401663136
4	6.468168543
5	6.462078248
6	6.878547557
7	6.435163716
8	6.57930126
9	6.540485362
Average	6.493692152
Agent System Average Error	6.447204044
Min Error	6.317212682
Max Error	6.878547557

Table 7.13: Average Error for Agent System using bagging. The average error is provided for each agent in the agent system, as well as the average of these values (Average of averages). The average error for the complete AgentSystem as a whole is shown as well.

agents conforming the system is indicated, as well as the average error of the complete agent system. The point of interest is comparing the average error of the agent system against the average of each agent's average error. The table with complete results for the case of an agent system with 10 agents is shown in table 7.13.

Table 7.13 shows that the Average error of the complete agent system is less than the average of the error of each agent. Some agents though perform better or worse by themselves. This is a consequence of the ability of ensemble systems to overcome the computational problem. Through having several agents, we ensure that we will not have the worst case and that in average we will perform better than a single agent. Final results for the case of 20 and 50 agents are shown in table 7.14. The same behavior observed with 10 agents is present in both cases.

From this experiment, we gathered information regarding the advantages of an ensemble on datasets where the computational problem may be present. By having mul-

20 Agents	
Average Error	6.494678716
Agent System Average Error	6.43324119
Min Error	6.305279274
Max Error	6.69750303
50 Agents	
Average Error	6.525559548
Agent System Average Error	6.445464955
Min Error	6.348826857
Max Error	6.873038798

Table 7.14: Average Error for Agent System. Results for the case of 20 and 50 agents.

tiple agents, we counter the cases where one agent makes a bad decision by multiple agents which predict correctly, and hence we reduce the risk of selecting a local optimum and have a performance better than a single agent can provide us in average.

7.5. Result analysis on the HLA multi-classifier

As described in chapter 5, the HLA dataset presents a novel problem which has not been attempted to be resolved through machine learning techniques. In our research, we modeled the problem as a multi-class classification problem and proposed a solution similar in nature to the solution applied to handwritten digit recognition. The HLA Dataset contains 23 different classes among which 59229 samples are divided. The number of positives for a certain class varies considerably from class to class, being that, for instance, class S22 has 3646 positives, while class 231 only has one. Each sample has 527 binary-valued features, where each feature represents the result to a specific probe. 0 indicates no reaction while 1 indicates reaction. Hence, the list of features indicates for which probes the HLA family did react and for which probes it didn't.

The goal of this experiment was to generate a good multi-classifier for the HLA

dataset, which was able to classify correctly at least a 95 % of samples. To achieve this, binary classifiers were built for each one of the 231 classes using adaboost*. In the end, a multi-classifier was built, where the prediction of the agent system is the corresponding class of that binary classifier with the highest prediction value. It is important to stand out that the case where all classifiers return a negative prediction, which means that all classifiers do not consider that sample to belong to their class, is possible. However, in this case we still return a predicted class, given by that binary classifier whose prediction is closer to 0. Its important to remember that the further a prediction is from the separation edge (which in this case is the value 0), the most confident the prediction is, so always taking the maximum prediction of all binary classifiers will consistently return us the class for which our prediction has the most confidence.

The preprocess methodology used was hard partition, as time constraints and not clear advantage of soft partition in previous experiments showed us it was the best option. This resulted in having 527 agents to choose from, each agent corresponding to one probe. This partition procedure was useful as well, as it allows us to make analysis also on the probe level.

Adaboost* was used as the learning algorithm, as it showed advantages over Adaboost on previous experiments. Due to time constraints, an upper limit of 100 iterations was set, and less iterations could be run in the case where the binary classifier achieved convergence (all samples in dataset correctly classified).

General results are presented in table 7.15. The multi-classifier achieved a generalization error of 3.007 %, which was a satisfactory result since only 1729 samples of 59229 are incorrectly classified.

The point of interest is now to analyze for which class it performed best, for which class it performed worst, and whether different sizes of positives in a class had an

Correct Classification:	57500
Incorrect Classification:	1729
Generalization Error:	3.0069565 %

Table 7.15: Overall results for HLA multi-classifier

	BER	AUC	BER Error	Precision	Recall	Total Positives
S1	0	1	0	1	1	91
S21	0	1	0	1	1	13
S112	0	1	0	1	1	15
S126	0	1	0	1	1	5
S141	0	1	0	1	1	120
S142	0	1	0	1	1	210
S153	0	1	0	1	1	15
S165	0	1	0	1	1	14
S231	0	1	0	1	1	1

Table 7.16: Results for binary classifiers which reached convergence

effect in the result. Table 7.16 shows results for those classes where convergence was achieved, and figure 7.10 shows the gamma progression for classes S21, S112 and S165, all of which achieved convergence under 100 iterations. These classes share the following characteristics:

- The number of positives for the class is relatively small, being class S142 the one with most positives (210) and class S231 with the least (1).
- The initial value for gamma is considerably high, as shown in figure 7.10. Also, the progression of gamma through the iterations does not fall, which indicates that the error of the best agent on the reweighted dataset does not increase considerably, and agents are able to continue contributing on reducing the misclassifications.

These observations provide us the following insights. The fact that gamma initially had a considerably high value, close to one, indicates that it was easy to classify as negatives most of the true negatives, which represent the great majority (over 59000

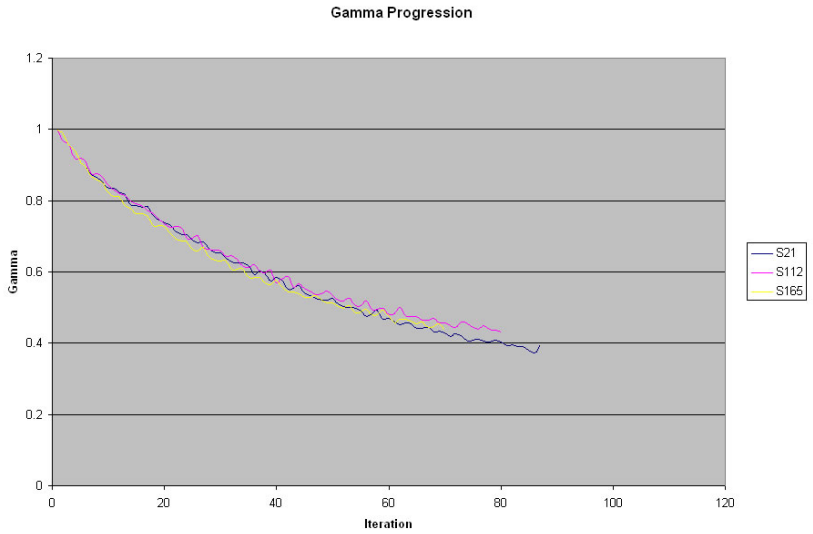

Figure 7.10: Graphical comparison of gamma progression for binary classifiers for classes S21,S126 and S165

samples are negative in all these examples, being in the best case only 210 positive). Although none of these classifiers reached 100 iterations, it is notable that the reduction of gamma still is important, and hence the margin can still be increased to increment confidence on predictions.

Table 7.17 provides us results for binary classifiers where the achieved precision was 0. This means, that the classifier was unable to achieve at least one true positive prediction. In these classes, boosting was unable to successfully separate samples which belong to a certain class from samples which do not. Figure 7.11 shows the gamma progression for the case of S16, S138 and S208. From this data, we may make the following observations:

- The groups where precision was 0 are of small size, having the biggest one 85 (S39) members an the smallest one 1 (214,216,226,228).
- The progression of alpha has a steep fall in the first iterations. This can be seen more clearly in figure 7.12 which contains graphs for classes with precision 1 and precision

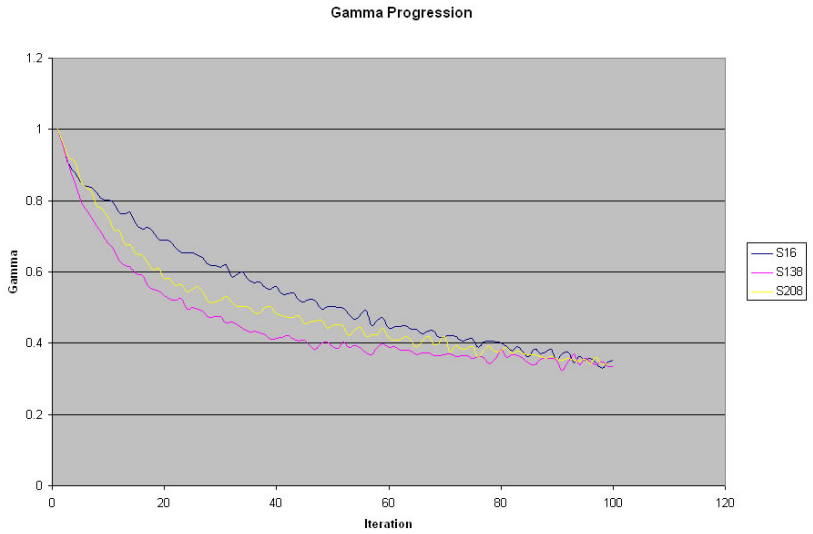

Figure 7.11: Graphical comparison of gamma progression for binary classifiers for classes S16, S138 and S208

0.

From these observations, given the steeper fall of alpha compared to other groups, we may conclude that agents in the first iterations, which are the ones which have the greater relevance for the ensemble. Given that gamma has an initial high value, and that initially the classifier takes care of samples which do not belong to the class, as they represent the vast majority, the steep fall indicates us that there were no agents which could successfully differentiate positives from negatives without considering a considerable amount of negative positives. This translates in that, considering the complete dataset, no probe could differentiate that group from other groups good enough to build a classifier. It is possible that considering only a subset of the dataset differentiation is possible, but for the complete dataset it was not.

Figure 7.13 shows a picture of the confusion matrix. In it, any position with a value different from 0 is highlighted with green background. An optimal confusion matrix only has elements on the main diagonal. In our matrix, we can see that, as our 3 %

	BER	AUC	BER Error	Precision	Recall	Total Positives
S16	0.5	0.5	26	0	0	13
S19	0.5	0.5	26	0	0	13
S39	0.5	0.5	170	0	0	85
S58	0.5	0.5	38	0	0	19
S76	0.5	0.5	50	0	0	25
S90	0.5	0.5	22	0	0	11
S93	0.5	0.5	22	0	0	11
S109	0.5	0.5	98	0	0	49
S124	0.5	0.5	10	0	0	5
S138	0.5	0.5	48	0	0	24
S148	0.5	0.5	30	0	0	15
S151	0.5	0.5	30	0	0	15
S163	0.5	0.5	28	0	0	14
S171	0.5	0.5	24	0	0	12
S174	0.5	0.5	24	0	0	12
S181	0.5	0.5	18	0	0	9
S184	0.5	0.5	18	0	0	9
S190	0.5	0.5	14	0	0	7
S193	0.5	0.5	14	0	0	7
S201	0.5	0.5	12	0	0	6
S205	0.5	0.5	8	0	0	4
S208	0.5	0.5	8	0	0	4
S211	0.5	0.5	2	0	0	1
S213	0.5	0.5	60	0	0	30
S214	0.5	0.5	2	0	0	1
S215	0.5	0.5	18	0	0	9
S216	0.5	0.5	2	0	0	1
S219	0.5	0.5	8	0	0	4
S223	0.5	0.5	60	0	0	30
S226	0.5	0.5	2	0	0	1
S227	0.5	0.5	18	0	0	9
S228	0.5	0.5	2	0	0	1

Table 7.17: Results for binary classifiers with precision 0

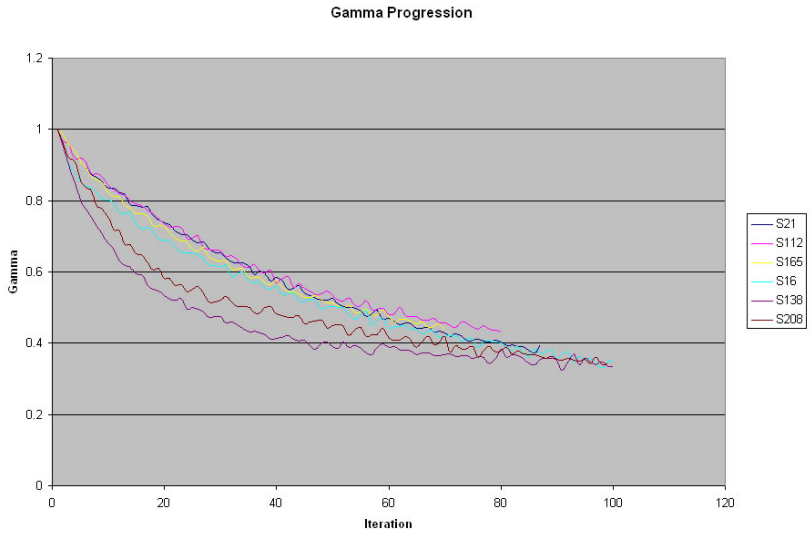

Figure 7.12: Graphical comparison of gamma progression for binary classifiers for classes S21,S126 and S165 (precision 1); S16, S138 and S208 (precision 0)

generalization error hinted us, the proportion of cells with value different of 0 not in the main diagonal is small. However, it is noticeable that the line crossing the diagonal is not continuous, and has white cells on its way. These cells represent those classes whose classifiers had 0 precision. Figure 7.14 shows a window of the confusion matrix, which shows that no element in S16 was classified as such, but all of them were classified as S8. This is a consequence of having a poor binary classified for S16. From the results, it is clear that the binary classifier for S8 gave the highest prediction value for all elements in the S16 group, which means that given the inability of the binary classifier of S16 to provide a positive value, from all the different class models represented by the binary classifiers, samples belonging to S16 are most similar to samples belonging to S8.

7.5.1. HLA probe results

For the complete classifier, 354 different probes were used, having only 220 an α value greater than 1. Table 7.18 shows the top 10 probes based on their cumulative

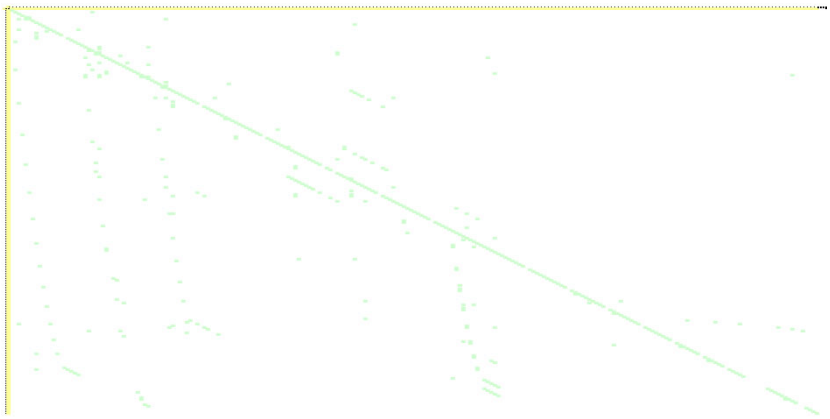

Figure 7.13: Zoomed out view of confusion matrix for HLA multi-classifier

	SUM	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18
SUM		91	1110	284	349	152	624	70	364	209	186	169	105	93	80	52	1	42	401
S1	91	91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S2	1105	0	1104	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S3	247	0	0	247	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S4	325	0	0	0	325	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S5	143	0	0	0	0	142	1	0	0	0	0	0	0	0	0	0	0	0	0
S6	636	0	0	7	0	8	620	0	0	0	0	0	0	0	0	0	0	0	0
S7	65	0	0	0	0	0	0	65	0	0	0	0	0	0	0	0	0	0	0
S8	312	0	0	0	0	0	0	0	312	0	0	0	0	0	0	0	0	0	0
S9	195	0	0	0	0	0	0	0	0	194	0	0	0	0	0	0	0	0	0
S10	182	0	0	0	0	0	0	0	0	0	182	0	0	0	0	0	0	0	0
S11	156	0	0	0	0	0	0	0	0	0	0	156	0	0	0	0	0	0	0
S12	117	0	0	12	0	0	0	0	0	0	0	0	98	0	0	0	0	0	0
S13	91	0	0	0	0	0	0	0	0	0	0	1	0	90	0	0	0	0	0
S14	78	0	0	0	0	0	0	0	2	0	0	0	0	0	76	0	0	0	0
S15	52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52	0	0	0
S16	13	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	0	0	0
S17	52	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	0	39	0
S18	390	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	390

Figure 7.14: HLA confusion matrix. Upper Left corner of matrix.

Number	ID	Total Alpha Sum	Times Chosen
1	74	352.7079997	229
2	487	140.0451577	216
3	81	63.6980766	22
4	427	43.79259383	40
5	439	35.47220656	20
6	28	34.90599983	21
7	462	27.43947088	21
8	142	26.76963259	72
9	290	25.60437399	41
10	446	25.01842776	59

Table 7.18: Top 10 probes based on cumulative alpha sum

alpha value among all classifiers. Interesting in this list are the first two probes with id 74 and 487. The first one is chosen by almost every binary classifier at the beginning of the boosting process. The reasoning for this is as follows: In the first iterations of boosting, as all samples share similar weight, the number of negatives considerably outnumber the number of positives, and hence it is more important to classify them as negatives than to classify the positives correctly. Probe 74 and 487 have the same result for almost all samples, which means that they are useful in the beginning to move most samples to the negative side of the separation margin. After the weights of the positives increase enough to counter the great number of negatives, it is then when probes are chosen which can classify such positives correctly.

7.5.2. Using multi-agent systems to model knowledge: the case of HLA

For each binary classifier we built, we chose those agents which in conjunction best separated samples that belonged to that class from samples that did not. The final set of agents with their associated weights is our model of that class. We use this model to measure whether another HLA, represented by its responses to probes, is similar to

the class in question.

When making a prediction, we use a weighted sum of all the agents considered in the classifier. The weight of an agent indicates its relevance and how much its decision affects the final decision. When all agents agree, the final result has a larger absolute value, which as mentioned before, translates into greater confidence.

A binary classifier allows us to indicate whether an HLA belongs to a group or not. But if we have two classifiers, how can we tell whether the classes they are classifying are similar or not?. This is achieved by looking at the agents and weights for both classifiers and look at the intersection. By intersection, we mean what portion of the final decision of both classifiers is always equal, and this is defined by the set of agents that exist in both classifiers and the minimum weight for each of those agents in both classifier. Figure 7.15 shows the intersection of the S8 and S16 classifiers. In it, the list of agents which exist on both classifiers is listed, as well as the minimum weight. The sum of the weight indicates the portion of the final response that will be equivalent for both, which taking in consideration the total sum of α for both S8 and S16 classifiers, represents the 47.86 % and 39.8 %. What these values mean, is that 47.86 % of the S8 classifier responses will be the same as 39.8 % of the S16 classifier.

In the confusion matrix we observed how S16 was confused by S8, given that the precision of the S16 classifier is 0. As the classifier for S8 shares almost 50 % of the model that indicates us what should be S16, it is expectable that as the S16 classifier was unable to learn correctly, S16 samples will receive relative high positive value when using the S8 classifier.

An important consideration is that based on the probes we chose, our class model, these two classes are similar. If only S8 and S16 were in the dataset, most likely we would have chosen different probes and they wouldn't be confused. It was observed that there were probes which were able to distinguish these 2 groups, but were not used

INTERSECTION				
	Alpha			
ID	S8	S16	Sign	Min Alpha
3	0.203892	0.215036	1	0.203892315
26	0.071578	0.077068	1	0.071578246
74	1.116363	2.347982	-1	1.116362735
130	0.225409	0.217539	1	0.217538575
142	0.146733	0.206871	-1	0.146732616
192	0.221213	0.208032	1	0.208031609
246	1.258043	0.973449	1	0.973449492
392	0.159812	0.249609	1	0.159811509
487	0.972046	1.674211	-1	0.972046458
525	0.513431	0.188313	1	0.188313144
Alpha Sum	8.896633	10.69907		4.257756698
Weight Percentage	47.86%	39.80%		

Figure 7.15: Intersection calculation for S8 and S16 classifiers

since they were not as useful when taking consideration of the complete dataset.

Figure 7.16 shows the intersection matrix. In this matrix, cell (i,j) indicates the intersection percentage of class i and class j based on the total alpha sum of class i. Cells in yellow have an intersection higher than the 43 %. Cells in green have an intersection lower than 10 %. What is interesting about this graph are the several visible patterns. Zones with high intersection have been indicated through squares. These zones revolve around the main diagonal, which indicates that a group of classes, close to each other in the numeration, have a high intersection based on our class models. A second interesting observation is that from group 22 to 34, there is very little intersection between classes, which is made clear by the green horizontal and vertical bars. Another interesting factor is that there seems to be a periodicity regarding classes which are similar to other classes, which in the diagram are represented by parallel diagonal lines. What these line indicate is that, for instance, S97 is similar to S7, S27, S46, S64, S81..., while S98 is similar to S8, S28, S47, S65, S82... and S99 is similar to S9, S29, S48, S66, S83...

Another interesting pattern seen in this matrix are parabola-shaped lines below the

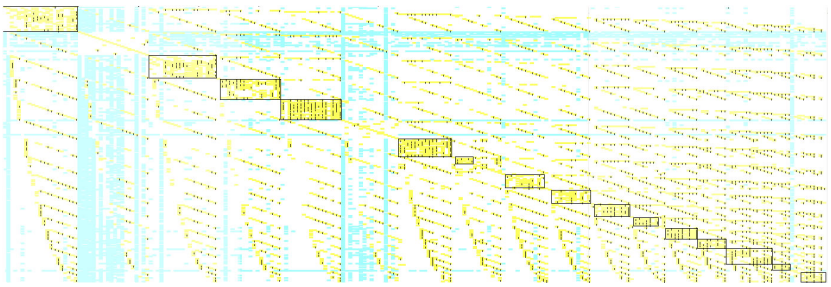

Figure 7.16: Zoomed out view of intersection matrix. Yellow cells indicate intersection of more than 43 %. Green cells indicate intersection of less than 10 %.

main diagonal. The existence of this pattern means that one class is similar to a set of consecutive classes, then the next class is similar to another set of consecutive classes placed below and so on. For instance, S5 is similar to S80 - S95, S7 is similar to S112-S126, S8 to almost all classes in S127-S134, S9 to S146-S150, S10 to S156-S159, S11 to S167-S176, S12 to S178-S183 and so on.

We have indicated a concept of similarity between classes indicated by the proportion of probes and alpha values common to both classes when building the binary classifier. As indicated before, this does not necessarily indicate that the groups are similar in a different context. What it does indicate is that a certain percentage of those probes which help us identify one class against all other classes are shared by both classifiers.

7.5.3. Using HLA multi-classifier

Another important aspect of this experiment was testing the export capability of the mage framework by exporting the HLA multi-class classifier and having it automatically rebuilt in a second application. This shows the automation capabilities of the framework, where after defining the components to be used while building the multi-agent system, it is generated, exported into an XML document and available for any other application to be used.

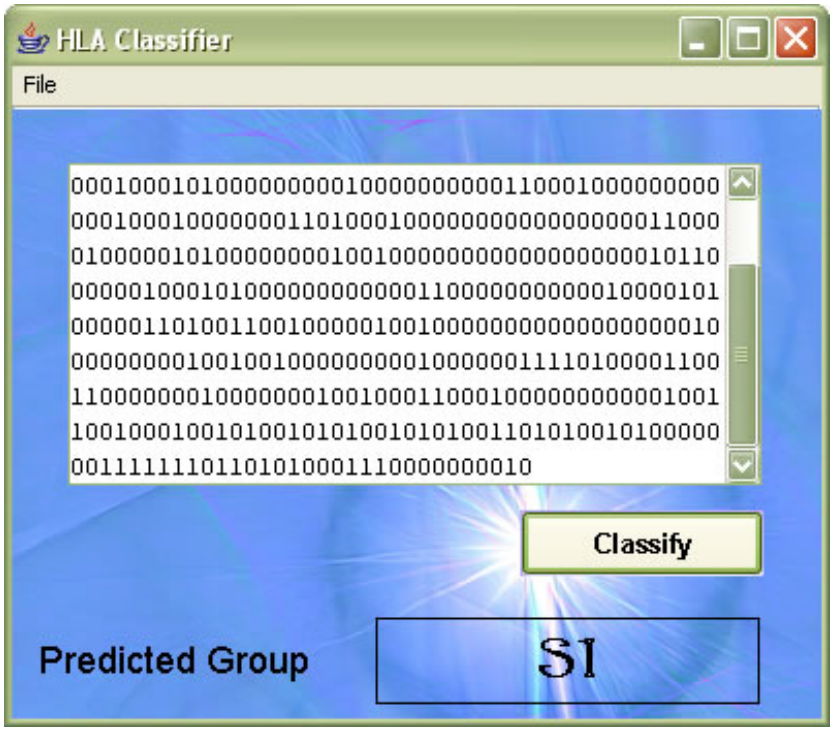

Figure 7.17: Screen shot of external application making use of multi-classifier.

A simple application was built which used the HLA classifier for both available and unseen probe answer sequences. Figure 7.17 shows an screen shot of the application. Using the multi-classifier on this external application was simple as we made use of the deserialize method of the AgentSystem class. However, any implementation could just deserialize the xml component, and as long as it has its own implementation of the required Resolver, Partitioner and Evaluator components it will be fairly easy.