

# Learning Sets of Sub-Models for Spatio-Temporal Prediction

Andrew Bennett and Derek Magee  
School of Computing, University of Leeds  
Leeds, UK, LS2 9JT  
{andrewb,drm}@comp.leeds.ac.uk

## Abstract

In this paper we describe a novel technique which implements a spatio-temporal model as a set of sub-models based on first order logic. These sub-models model different, typically independent, parts of the dataset; for example different spatio or temporal contexts. To decide which sub-models to use in different situations a context chooser is used. By separating the sub-models from where they are applied allows greater flexibility for the overall model. The sub-models are learnt using an evolutionary technique called Genetic Programming. The method has been applied to spatio-temporal data. This includes learning the rules of snap by observation, learning the rules of a traffic light sequence, and finally predicting a person's course through a network of CCTV cameras.

## 1 Introduction

Events over time may be described using a spatio-temporal data description. In all but the simplest events this may involve multiple (variable number of) objects; and multiple spatial, and temporal contexts. A description, and any model based on this description, must support this complexity. Our novel technique learns a set of sub-models that model different, typically independent, aspects of the data. To combine the sub-models into a single model a context chooser is used. This picks the most appropriate set of sub-models to predict in a certain context.

We have applied the technique to learning games by observation, and also for predicting how objects will move through a network of CCTV cameras. Both of these approaches benefit from breaking the solution down into its component parts, and modelling each part separately. In the game playing scenario this typically means finding a sub-model for each of the outcomes in the game. In the CCTV scenario this means learning the actions of single objects moving in the scene.

Global models typically will only be able to predict from data which has an exactly similar input structure to the training data. For example this means that if it was trained on a dataset containing single actions, it would fail to predict on a dataset containing multiple actions. This is because the system has not seen any data in this combined form, and may not be able to predict it. However, our technique can work in this scenario, because instead of having to

find a model that matches all the combinations of actions, each of the actions occurring at the same time will be modelled by a separate sub-model, and the resulting output can simply be combined. The sub-model learning process has feature selection implicitly built in, so the learner can not only find and model the key actions in the dataset, but it also uses the most appropriate data items to detect and predict them. This can often simplify the models to be learnt.

There has been much previous work on learning from spatio-temporal domains. Traditional methods like decision trees, neural networks and support vector machines (SVMs) require a fixed length vector to represent the world. To construct this vector often requires knowledge of the domain, making these methods hard to use in a problem domain where the structure of the domain is variable, and not known a priori. In particular, only a subset of the data may be relevant, with the rest acting as a distractor. Feature selection is used to find this subset, which then allows for a more general model to be built. However, the relevant subset may change from one context to another. To perform feature selection, a set of variables is selected from the input set, and a goodness value is computed by using an objective function. By looking at the change in the function's value variables can be added or deleted. This will then repeat until convergence. An overview of feature selection can be found in [8]. One approach to modelling data of variable length is to take statistics of a variable size set [21] and [7]. This produces a fixed set description suitable for use with SVMs etc. However, spatial relationship information is lost in this process. If this information is important within a domain this leads to a poor model.

Temporal modelling approaches such as Markov chains, Hidden Markov Models (HMMs) [18] and Variable Length Markov Models (VLMMs) [6] use a description based on graphs to model state transitions. These methods still need a fixed size input vector, but can optimise their structure by using local optimisation approaches based on information theory [1]. In VLMMs this optimisation acts as kind of temporal feature selection, but as the input variables stay in the same fixed order spatial feature selection is not performed.

Bayesian networks are a generalisation of probabilistic graph based reasoning methods like HMMs and VLMMs. Again these networks require a fixed input vector, but again their relational structure can be optimised by local search [13], genetic algorithms [4], or MCMC [5] usually based on information theoretic criteria.

An alternative to using graph based methods is to use (1st order) logical expressions. Feature selection is implicit in the formalism of these expressions thus removing the need to use a fixed length vector. Logical expressions also make no assumptions about the ordering of variables, so there is no need to have a have them in a fixed ordering. Progol [15] and HR [2] are Inductive Logic Programming (ILP) methods. ILP in general takes data and generates a set of logical expressions describing the structure of the data. Progol does this by iterative subsumption using a deterministic search with the goal of data compression. HR does this by using a stochastic search using a number of specialist operators. These approaches suffer from a number of disadvantages.

Firstly, logical expressions are deterministic, so it is hard for then to model non-deterministic situations. However, there has been much work on combining (1st order) logic and probability to solve this problem [17] and [9]. Secondly Progol's search is depth bounded, which limits the size of problems it can work on, as explained in [16]. Thirdly Progol uses untyped data, this allows nonsensical models to be produced, and also increases the search time to find good models. Fourthly Progol only uses Horn clauses, which make it hard to represent some kind of logical expressions. Finally Progol's fitness function is only based on how well the model compresses the data, and not how well the model predicts the data. This can cause incorrect, or invalid models to be produced. [19] present a comparison of Progol and HR on a cognitive vision task, concluding that Progol performs better than HR. HR was produced to solve mathematical problems, so this probably shows that its representation, and operators are not well suited in a cognitive vision context. However, this work also shows that the performance of Progol is also limited.

Genetic Programming (GP) [10] is a evolutionary method, similar to genetic algorithms, for creating a program that model a dataset. In a similar way to HR, it takes a dataset data, a set of terminals, and a set of functions; and using a set of operators generates a binary tree that models the data.

There has been recent work on trying to build predictive models of basic games, using unsupervised approaches. [16] produced a system that could learn basic card games. It had three parts: an attention mechanism, unsupervised low-level learning, and high-level protocol learning. The attention mechanism uses a generic blob tracker, that locates the position of the moving objects. From this a set of features including: colour, position and texture are extracted. Clustering is used to cluster the data into groups. Using these clusters new input data is assigned its closest cluster prototype. A symbolic data stream is then created by combining together the clustered data, with time information. The symbolic stream is passed to Progol, which builds a model of the data. Once the model has been learnt it can be applied to new data. This allows the system to interact in the world.

Other related work in this area [3] looked at training a computer to play a peg-hole game. The game area was represented as an attributed relational graph. The vertices represent the objects in the scene, and the edges represent the 3D distance between a pair of objects. A percept distance metric is used to compare the similarity of two different game graphs. Using this metric a hierarchy can be founds that shows similar game playing scenarios across a dataset. Key scenes can then be extracted from the game data. These are typically when an object is moved or put down. To find the most appropriate action a particle filter is used. Initially the a set of hypothesis are set to the same (current) state, the actions are created by randomly sampling the set of actions that could cause the current state. Once the filter is initialised the actions are applied to the hypothesis, and the distance from the new state is taken off from the solved state. The worst hypotheses are then removed. Finally the next set of actions is calculated and the process repeats. After a number of iterations the optimal state is found.

[20] looked at learning event definitions from video. A raw video of a scene is converted into a polygon representation. This is then transformed into a force-dynamic model which shows how the objects in the scene are in contact with one another. Using this data and-meets-and (AMA) logic formulae describing the events are learnt using a specific-to-general ILP approach.

The remainder of this paper will take the following form. The second section talks about the architecture for the models. The subsequent section describes how these models are learnt by Genetic Programming. The subsequent section presents an evaluation of our system, and the final section shows the conclusions of the work and the further work.

## 2 Architecture for Models of Spatio-Temporal Data

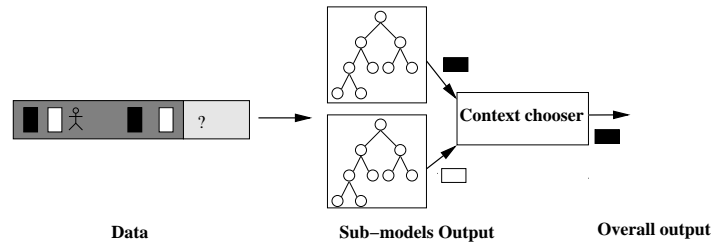


Figure 1: This figure shows the architecture of our model. It has two parts: a set of sub-models, and a context chooser to decide how to use the sub-models in different situations.

Figure 1 shows the architecture of our model. It is broken down into two parts: the sub-models, and the context chooser. The sub-models each model a separate part of the underlying process generating the data. Each sub-model contains two sections: a search section, and an output section. The search section looks for a particular pattern in the dataset. A query language, which has some similarity to SQL and Prolog, is used to describe the actual search, and a binary tree is used to represent it. The output section describes what is implied if the search returns true. This will be a set of entities and their properties the sub-model predicts will happen next. Figure 2 shows an example of a sub-model. The functions are standard logic functions: And, Or, and Not; as well as equally operators: Equals and Not Equals.

The context chooser is used to decide how to combine the sub-models in different situations. It takes as its input a boolean vector showing which sub-models have returned outputs, and makes a decision as to which ones will form the overall output. There are currently two kinds of chooser: a deterministic chooser, and a probabilistic chooser. The following sections will explain about them.

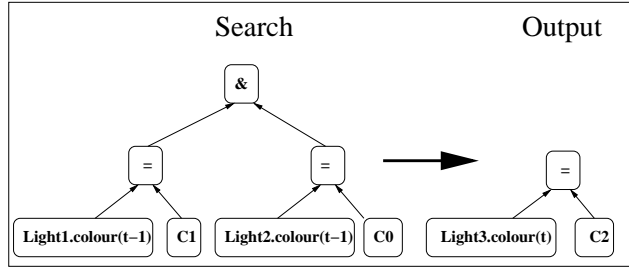


Figure 2: This shows a sub-model matching a traffic light with colour c1, and a different light having a colour c0 both at current time - 1. If the expression evaluates true it will output a new light which has a colour c2, at the current time.

## 2.1 Deterministic Chooser

The deterministic chooser is a transparent chooser. It simply returns the union of the outputs from all the sub-models that have evaluated true. It is only really useful for modelling deterministic datasets.

## 2.2 Probabilistic Chooser

The probabilistic chooser uses a probability distribution to decide what happens when a certain sub-set of sub-models produces an output. It can therefore be used to model stochastic datasets, where a particular input pattern can result in multiple possible outputs. It can also be applied to deterministic datasets giving potentially better results than using the deterministic chooser. With the deterministic chooser the sub-models must be modelling separate parts of the dataset, and interaction is limited. The probabilistic chooser works in a different way: a sub-model can be found that models a global concept in the dataset, and smaller sub-models can then be evolved to cover the cases where this global concept breaks down. This means the learner can find much simpler sub-models compared to the ones used with the deterministic chooser. This is analogous to the if/else constructs found in high level computer languages.

A context  $S_n$  is defined as a set of sub-models  $M$  producing an output in a given context, for example  $S_n = M_1, M_2$  represents that  $M_1$ , and  $M_2$  have produced an output at the same time. For each context a probability distribution over the overall output is defined for example  $P_n(M_1), P_n(M_2), P_n(M_1, M_2)$ . This distribution is formed from the frequency of occurrence of each situation in the training data in the given context. This then can be implemented as a sparse hash table.

## 2.3 Complexity of the problem

Every sub-model is stored as a binary tree. Assuming our query language has  $N_s$  functions and constants in it, and the number of nodes in the tree is  $N_n$  the

complexity of generating every possible tree is:  $O(N_s^{N_n})$ . Assuming  $N_n = 10$  the number of trees which can be generated at with depth 2 is  $10^{2^2+1}$ , or  $10^3$ . If we increase the depth to 3 the number of unique trees becomes  $10^{2^3+1}$ , or  $10^7$ , this is an increase of  $10^4$  for just one increase in the depth of the tree. We ran a test generating all possible trees of depth 2 for the Snap dataset, on a standard desktop machine which took 13s. Assuming the increase, generating all possible trees of depth 3 is  $13 * 10^4$ s, which works out to be more than 1 day of processing time. Once this has been completed there still leaves the problem of what subset of sub-models needs to be used to finding the optimal overall model. This clearly makes the problem of finding the optimal set of sub-models for anything other than a simple dataset an intractable problem to solve by exhaustive search. It should be noted that the complexity of searching for a global solution in a neural network by exhaustive search is similar to the problem just described. However the use of local search (an approximate method) makes the optimisation of the problem tractable.

### 3 Learning the Models from Data

As was shown in the previous section it is intractable to find the set of optimal sub-models by exhaustive search therefore an approximate search strategy needs to be used. We have decided to use Genetic Programming (GP) to learn our sub-models. It was chosen for two reasons, firstly it has already been successfully used to learning computer programs for pattern recognition tasks[11]. Secondly we have a discrete domain and it is not clear how local search would work here. Using a stochastic search seems like a better solution.

Genetic Programming (GP) [10] evolves a population of programs until a program with the desired behaviour is found. It is a type of genetic algorithm, but the programs are stored as binary trees, and not as fixed length strings. Functions are used for the nodes, and terminals (for example constants, and variables) are used for the leaf nodes. In order for the population to evolve a fitness function (in our case an accuracy score) must be defined. This score will be used by the GP system to decide which programs in the current generation to use to produce the next generation, and which ones to throw away. To initialise the system, a set of randomly generated programs must be created. Each then receive a score using the fitness function. Algorithms including crossover, mutation and reproduction use the programs from the current generation to create a new generation. Crossover takes two programs and randomly picks a sub-tree on each program, these two trees are swapped over, creating two new programs. Mutation takes one program, randomly picks a sub-tree on it, and replaces it with a randomly generated sub-tree. Reproduction copies a program exactly as it is into the new generation. The programs in the new generation are then scored, and the process is repeated. The GP system will stop when a certain fitness score is reached, or a certain number of generations has passed.

In GP it is assumed that every non-terminal will have closure. This means it will accept any type of data from any other terminal or non-terminal. For

many problems this is an unrealistic assumption to make. [14] adds typing to the GP creation, and modification processes. This means that terminals have a data-type, and non-terminals have argument, and return data-types. The use of data-types puts restrictions on how trees can be created, and evolved. Montana works out for each non-terminal what terminals and non-terminals can be placed below it, so that the tree will still be valid. This prevents invalid trees being created. When crossover is performed checks are made to ensure that the types of each of the nodes are the same, to preserve the validity of the trees. We use GP with data typing for our system. In standard GP all the program code is placed in one main function. Normally when humans are writing code they make use of functions to break the code into re-usable sections. This approach was applied by Koza to create automatically defined functions (ADFs) [11]. Here functions were added to the GP trees. Crossover, and mutation can still be applied to evolve the trees, but structure preserving crossover is used to ensure that the new trees will still remain valid. Later work added the ability to add, duplicate, and delete functions, which allowed the GP system to specialise, and generalise, functions [12].

We also apply a structure to our trees to try and cut down the search space, and to make finding a solution more efficient. Our architecture can be seen as a ADF where the ADFs are replaced by the sub-models, and the main program by the context chooser. We, however, don't use all the methods from [11] to evolve our model because the ADF system is essentially still evolving single global models, and we need to evolve separate sub-models. Also, the basic structure of the chooser does not need to be evolved, we have a closed form solution that does not need refinement. Some of methods used to evolve ADFs evolve the structure of the solution, using these would make our search less efficient.

To initialise the population we generate a set of models just containing one randomly generated sub-model. The sub-model is produced using Koza's ramped half and half method [10]. We investigated creating an initial population with models containing multiple sub-models, however this approach did not perform well in ad-hoc tests. The models are then scored using the fitness function described below. To prevent overfitting a moving window which is 80% of the size of the dataset is placed over the dataset. At the start of each new generation the window is moved to a new random location. Next a new generation produced using the operators below. The learner will stop when a model of score 0 (this only will occur in non-noisy training sets) is produced, or it exceeds the maximum number of generations.

### 3.1 Operators

The system uses two kinds of operators. Firstly there are operators that try to optimise the sub-models used in the model, and secondly there are operators that optimise the sub-models themselves. A technique called tournament selection [10] is used to pick a model from the population. Tournament selection picks  $n$  models at random from the population, and returns the one with the

lowest score, for our experiments we set  $n$  to be 10. The operators used to optimise the sub-models used in the model are shown below:

**Reproduction** A set number of models are picked via tournament selection and copied directly into the new population. (This occurs 10% of the time.)

**Adding in a sub-model from another model** Two models are picked by tournament selection. A sub-model from the first picked model is randomly selected, and added to the second chosen model. (This occurs 5% of the time.)

**Replacing a sub-model** Again two models are picked by tournament selection, and a sub-model from the first chosen model is then replaced by a sub-model randomly selected from the second chosen model. (This occurs 5% of the time.)

The only operator used to optimise the sub-models themselves is crossover. In crossover two models are picked using tournament selection. A sub-model from each model is then randomly selected, and standard crossover [10] is performed on these sub-models. (Crossover is performed 80% of the time.)

### 3.2 Fitness Function

To decide the fitness of a model ( $m$ ), its prediction error ( $E$ ), and coverage ( $C$ ) over the dataset is computed. For each time point in the dataset the model is given a set of history data ( $h$ ) and is queried to produce a prediction. This is then compared with the observed data ( $r$ ). Coverage is the absolute difference between the number of data items (*Elements*) produced by the model, and the number of items in the observed data. The coverage score then acts as a penalty for overfitting or underfitting a part of the dataset. To compute the error the difference between the result returned by the model and the actual result is calculated. A penalisation penalty ( $p$ ) is used as a penalty for models that incorrectly fire in the wrong part of the dataset. The overall score ( $S$ ) is then the addition of the accuracy, and the coverage. This is then repeated over the rest of the dataset, and the results are summed together.

$$C(r, h) = Elements(m(h)) - Elements(r) \quad (1)$$

$$E(r, h) = (m(h) - r) * p \quad (2)$$

$$S = \sum_i E(r_i, h_i) + C(r_i, h_i) \quad (3)$$

## 4 Evaluation

The learner was evaluated on three different datasets, which were: handcrafted traffic light data, handcrafted snap data, and data from people walking through a network of mock CCTV cameras. More detail about these datasets is presented in the following section.

## 4.1 CCTV Data of a Path

A 10 minute video of people walking along a path containing a junction was filmed. This was then used to mock up a network of CCTV cameras. Figure 3 shows a frame from the video. Virtual motion detectors, representing CCTV cameras, were hand placed over the video has shown in Figure 3. Using frame differencing, and morphological operations the video was processed to determine the location of the motion. If the number of moved pixels in a region exceeded a fixed threshold then the virtual detector outputted that motion had occurred at that location. Hysteresis on the motion detection is implemented as a 2 state, state machine (where the states are motion/no motion). The state machine requires a numbers of frames (normally 10) of stability to change state. The data produced is then placed in a datafile with a motion event recorded per state change going from no motion to motion. This datafile was split into a test file containing 30 state changes and a training file containing 50 state changes.

Figure 3 shows the main actions that occurred in the video. People either walked from region 1 through state 0 to state 3, and then back to region 1 again via region 0, or they again walked from region 1 through region 0 to region 3, but this time they walked to region 2 via region 0, finally the other possible action is to walk directly to region 2 from region 1 via region 0.

## 4.2 Traffic Lights

The traffic light file was handcrafted and complies to the standard UK traffic light sequence. We chose this as it has a variable number of lights lit at any one time. Three datasets were prepared: a non-noisy training set, a noisy training set, and a test set. The non-noisy training set and the test set were identical, and contained 200 state changes, and 250 objects on total. The noisy training set was created by adding 10% noise to the non-noisy training set. The noise involved adding, and changing the order of the lights in traffic light sequence.

## 4.3 Snap

The snap dataset was handcrafted, but the format of it was the same as the snap dataset used in the work of [16]. The snap sequence is the following:

$t_1$	$t_2$	$t_3$	$t_4$	...
BLANK	PLAY	CARD 1	RESULT	...
		CARD 2		...

Initially the computer will see a blank scene, then it will hear the word play, next two coloured cards will be seen. If they are the same then the word equals will be heard, otherwise different will be heard. We ask the computer to only learn the sections where a human is speaking, as it would be impossible to accurately predict the next two cards because they are essentially random. Again three datasets were prepared: a non-noisy, and noisy training set, and a test set. The datasets were all contained 200 state changes, and typically had

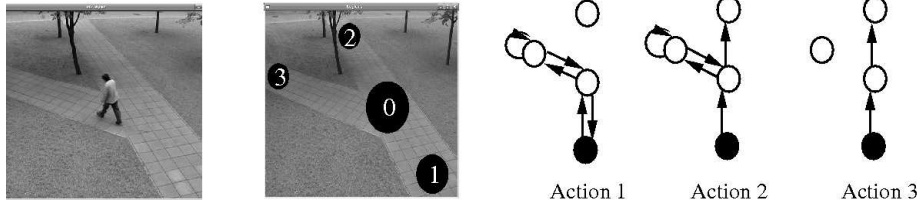


Figure 3: This figure firstly shows a frame of the video with a person taking a decision at the junction point, secondly it shows where the virtual detectors are on the video. Finally, it shows the three main actions in the video. Either people will walk from region 1 to region 3 and back again, or they will walk from region 1 to region 3 to region 2, or finally they can just walk directly from region 1 to region 2.

Dataset	Number of runs	Coverage(%)	Prediction Accuracy(%)
Snap No Noise	2	100	100
with a Deterministic Chooser	3	92.8	100
Snap Noise	2	100	100
with a Deterministic Chooser	1	94.4	100
	1	95.6	100
	1	88.4	100
Snap No Noise	5	100	100
with a Probabilistic Chooser			
Snap Noise	2	100	See Figure 6
with a Probabilistic Chooser	3	93	See Figure 6

Figure 4: This figure shows the results for the snap datasets.

250 objects in total in them. The noisy data was generated by adding 10% noise to the non-noisy training set. The noise took the form of removing cards, removing the play state, and changing the output state, for example making the output not equal when it should be equal.

## 5 Results

To test the system five runs were allocated to each possible combination of dataset, and chooser. For each run a different random number seed was used to initialise the system.

To evaluate how well the models have been learnt they were tested on a test set. Two metrics were used to evaluate the results: coverage, and predic-

Dataset	Number of runs	Coverage score(%)	Prediction accuracy(%)
CCTV	3	94	See Figure 6
	1	83	See Figure 6
	1	77	See Figure 6

Figure 5: This figure shows the results for the CCTV dataset

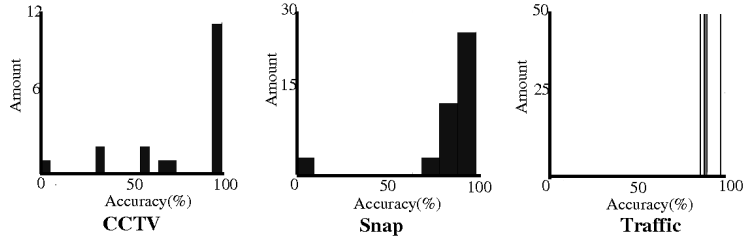


Figure 6: This shows the distribution of prediction accuracy for a typical model learnt for traffic, snap and path using noisy training data.

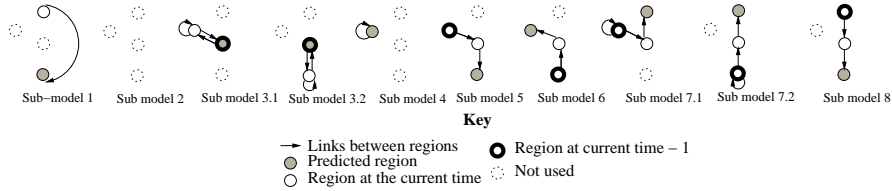


Figure 7: This figure shows the patterns matched for each sub-model from one of the results. Sub-models 3 and 7 have two possible outcomes. Note that the sub-models have different history lengths.

tion accuracy. Coverage scores if the system can correctly predict the dataset (ie. the probability of correct prediction is greater than 0%), and prediction accuracy scores with what probability the correct prediction is made. In non-deterministic scenarios this will not be 100%.

The system was initially tested on the traffic light dataset. Both the deterministic and the probabilistic choosers were used to learn from the noisy, and non-noisy training data. As this dataset is deterministic both of the choosers managed to learn all the non-noisy training data with 100% prediction accuracy, and 100% coverage. Almost all the results had three sub-models, with each sub-model modelling one of the lights. The deterministic chooser got 100% prediction accuracy and 100% coverage on all its runs from the noisy training set. Most of its results had multiple sub-models to model each different light. The probabilistic chooser did get 100% coverage, but it did not get 100% prediction accuracy, this is because the probability distributions modelled some of the noise in the training set. The typical prediction accuracy for one of the runs is shown in Figure 6.

Next, the system was tested on the snap dataset. Again both the probabilistic, and deterministic choosers were used to learn from the training sets. The deterministic chooser managed to get 100% prediction accuracy in both learning from noisy and non-noisy training sets. In the non-noisy set only two runs got 100% coverage, these both found separate sub-models for play, equal and not-equal. The runs for which the deterministic chooser did not get 100% coverage only had sub-models which only covered the not-equals and play states,

and they had failed to find a sub-model which covered the equals state. In the noisy training set the deterministic chooser again only got two runs to have 100% coverage. In the remaining runs the results were not general enough, and the sub-models had only learnt patterns occurring in the training set.

The probabilistic chooser managed to learn the non-noisy training set with 100% coverage and 100% accuracy. The sub-models learnt were slightly simpler than the results produced by the deterministic chooser. This is because most of the models had a sub-model that covered a general concept like non-equal, and when this concept broke down a simpler sub-model like equals would cover it. This was all modelled in the chooser probability distribution. When the probabilistic chooser learnt the noisy training data it only managed to get 100% coverage in two of the runs. The remaining runs like in the deterministic chooser found sub-models that were not sufficiently general enough. The typical prediction accuracy for one of the runs is shown in Figure 6

Finally the system was tested on the CCTV dataset. The results are shown in Figure 5, and the prediction accuracy in Figure 6. The dataset was only tested using the probabilistic chooser. Three out of the five results learnt every sub-model apart from one. From looking at the training set the missing sub-model appears to be not learnt due insufficient data. The actual sub-models learnt for one of the results are shown in a diagrammatic form in Figure 7. Looking at these you can see that the system has learnt small actions in the dataset, for example sub-model 6 shows the action of going to region 3 from region 1 (taking the left branch on the path). Figure 8 shows the probability distribution that decides how to use the sub-models. Two main parts of it are worth pointing out. Context 10 represents how the model deals with someone who has walked from region 1, to the junction point at region 0. The person can either go to region 3 (turn left) or region 2 (go straight on), and this entry models how likely they are to go to each region. Context 11 deals with a person who has already walked from region 1 to region 3, but now is coming back down the path and is at the junction point region 0 again. The two possible options are to go back to region 1 (turn right) or to go to region 2 (turn left), and again this entries models how likely each region is.

## 6 Conclusions

We have shown that that it is possible to learn a set of sub-models that can model different, typically disjoint, parts of a dataset. This technique is useful for a number of reasons. Firstly, it allows the system to learn from a dataset containing single actions, and then be able to predict from a dataset containing multiple overlapping actions. Secondly, model learning is easier and faster. If the dataset genuinely contains independent components then the complexity of the search is reduced to being linear in the number of contexts, rather than exponential (if every combination must be considered). Thirdly, being logical expressions, the sub-models are human readable, which makes them easy to check, and validate.

Context	Sub-Models giving Output	Output
1	sm3, sm4	sm4 (100%)
2	sm6	output nothing (50%), sm6(50%)
3	sm6, sm3	sm6 (100%)
4	sm6, sm1	sm3 (100%)
5	sm6, sm3, sm4	sm4 (100%)
6	sm3, sm4, sm5, sm6	sm4(100%)
7	sm7	output nothing (50%), sm7 (50%)
8	sm7, sm1	sm7 (100%)
9	sm4, sm7	sm4 (100%)
10	sm6, sm7	sm6 (25%), sm7 (75%)
11	sm5, sm7	sm5 (66%), sm7 (33%)
12	sm3, sm6, sm8	sm3 (100%)
13	sm6, sm7, sm8	sm8 (100%)

Figure 8: This shows the probability distribution for one of the results for the CCTV dataset.

The results show that using a probabilistic chooser performs better than a deterministic chooser, even in deterministic scenarios, because the probabilistic chooser can find a global sub-model that matches most of the dataset, and then learn simple sub-models to cover the cases where this global sub-model does not work. With the deterministic chooser the sub-models need to be more complex to represent the same scenario, and this makes the learning process harder, and slower.

In future work we will look into investigating more operators to improve the speed, and effectiveness of the GP learner. We are also looking into adding a compactness criterion to the learner, so that the models are of optimal size, and don't contain too many irrelevant terms. Finally we will be looking into using temporal, and spatio relations in both the datasets, and in the system.

## References

- [1] Matthew Brand. Pattern discovery via entropy minimization. In *Artificial Intelligence and Statistics*, 1998.
- [2] Simon Colton, Alan Bundy, and Toby Walsh. Automatic identification of mathematical concepts. In *International Conference on Machine Learning*, 2000.
- [3] Liam Ellis and Richard Bowden. A generalised exemplar approach to modeling perception action coupling. In *International Workshop on Semantic Knowledge in Computer Vision*, 2005.
- [4] R. Etzeberria, P. Larranaga, and J.M. Picaza. Analysis of the behaviour of genetic algorithms when learning bayesian network structure from data. *Pattern Recognition Letters*, 13:1269–1273, 1997.
- [5] Nir Friedman and Daphne Koller. Being bayesian about network structure. *Machine Learning*, 50:95–126, 2003.

- [6] Aphrodite Galata, Neil Johnson, and David Hogg. Learning behaviour models of human activities. In *British Machine Vision Conference (BMVC)*, 1999.
- [7] Kristen Grauman and Trevor Darrell. The pyramid match kernel:discriminative classification with sets of image features. In *International Conference on Computer Vision*, 2005.
- [8] Isabelle Guyon and Andre Elissee. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 2003.
- [9] R Haenni. Towards a unifying theory of logical and probabilistic reasoning. In *International Symposium on Imprecise Probabilities and Their Applications*, pages 193–202, 2005.
- [10] John Koza. *Genetic Programming*. MIT Press, 1992.
- [11] John Koza. *Genetic Programming II*. MIT Press, 1994.
- [12] John Koza, Forrest H Bennett III, David Andre, and Martin Keane. *Genetic Programming III*. Morgan Kaufmann, 1999.
- [13] Philippe Leray and Olivier Francios. Bayesian network structural learning and incomplete data. In *Adaptive Knowledge Representation and Reasoning*, 2005.
- [14] David Montana. Strongly typed genetic programming. In *Evolutionary Computation*, 1995.
- [15] S.H. Muggleton and J. Firth. CProgol4.4: a tutorial introduction. In *Relational Data Mining*, pages 160–188. Springer-Verlag, 2001.
- [16] Chris Needham, Paulo Santos, Derek Magee, Vincent Devin, David Hogg, and Anthony Cohn. Protocols from perceptual observations. *Artificial Intelligence*, 167:103–136, 2005.
- [17] N. J. Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–87, 1986.
- [18] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
- [19] Paulo Santos, Simon Colton, and Derek Magee. Predictive and descriptive approaches to learning game rules from vision data. In *Ibero-American Artificial Intelligence Conference*, 2006.
- [20] Jeffrey Mark Siskind. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *Artificial Intelligence Research*, 15:31–90, 2000.
- [21] C. Wallraven, B. Caputo, and A. Graf. Recognition with local features:the kernel recipe. In *International Conference on Computer Vision*, 2003.