Efficient Training Set Use For Blood Pressure Prediction in a Large Scale Learning Classifier System

Erik Hemberg
ALFA Group, MIT CSAIL
hembergerik@csail.mit.edu

Kalyan Veeramachaneni
ALFA Group, MIT CSAIL
kalyan@csail.mit.edu

Franck Dernoncourt
ALFA Group, MIT CSAIL
francky@csail.mit.edu

Mark Wagy
University of Vermont
mark@gmail.com

Una-May O'Reilly
ALFA Group, MIT CSAIL
unamay@csail.mit.edu

ABSTRACT

We define a machine learning problem to forecast arterial blood pressure. Our goal is to solve this problem with a large scale learning classifier system. Because learning classifiers systems are extremely computationally intensive and this problem's eventually large training set will be very costly to execute, we address how to use less of the training set while not negatively impacting learning accuracy. Our approach is to allow competition among solutions which have not been evaluated on the entire training set. The best of these solutions are then evaluated on more of the training set while their offspring start off being evaluated on less of the training set. To keep selection fair, we divide competing solutions according to how many training examples they have been tested on.

Categories and Subject Descriptors

F.1.1 [**Models of Computation**]: Genetics-Based Machine Learning, Learning Classifier Systems

General Terms

Algorithms, Performance

Keywords

Genetics-Based Machine Learning, Learning Classifier Systems, blood pressure prediction

1. INTRODUCTION

Large repositories of data offering the potential for inferential analysis via Machine Learning (ML) are becoming more ubiquitous. For example, in this contribution, we are referencing a large-scale medical database called MIMIC II[1] consisting of time series digital versions of physiological signals, and detailed clinical and bedside information from an Intensive Care Unit (ICU) setting. The first version of the publicly available data comprises roughly 18 physiological waveforms for around 10,000 patients and clinical information for about 40,000 patients. Physiological data alone is about 4TB. The broad goal of our current project is to knowledge mine the arterial blood pressure (ABP) waveforms for insights related to pattern recognition in beat sequences, prediction of ABP and classification of ABP characteristics. In this paper we present our first definition of a blood pressure prediction problem and offer preliminary performance results.

Because our knowledge mining system, EC-Star, (described in more details in Section 3), is a variant of a learning classifier system (LCS) [24] and because the ABP waveform data will eventually comprise many segments from many patients, we need to investigate approaches that will scale to a large set of training data. We are concerned that executing even one classifier ruleset against all the training data will be extremely costly and this factor will be multiplied upward when we use larger population sizes and complex representations which define large search spaces. In this contribution we explore an approach to efficient use of training data that cuts down on the amount of training cases used in preliminary evaluation of a candidate solution.

Our strategy allows competition among solutions which have not been evaluated on all the training set. The best of these solutions are then evaluated on more of the training set while their offspring start off being evaluated on less of the training set. To keep competition fair, we isolate competing solutions according to how many training examples they have been tested on. In studying our approach, we also show, in a very preliminary manner, how a scalable LCS system can be used to predict the arterial blood pressure of patients in intensive care units.

We proceed as follows: Section 2 describes our definition of a blood pressure prediction problem. Because our dataset is public, this definition can act as a benchmark for our research community. Section 3 briefly describes EC-Star. Within it, Section 3.5 presents work related to EC-Star and efficient training set use. Section 4 then provides more details and pseudocode of our approach. Section 5 evaluates

[1]http://mimic.physionet.org

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

the strategy in the context of solving the arterial blood pressure prediction problem. Section 6 concludes and describes future work.

2. ABP PREDICTION

In this contribution our goal is to predict the arterial blood pressure (ABP) of patients in an intensive care unit. Knowing whether a patient's arterial blood pressure will remain within normal function or lapse into a hypotensive (low) or hypertensive (high) regime in a short forward time window is valuable. In the ICU it allows staff adjustment, improved patient oversight and informs timely treatment.

To enable prediction, a dataset for a large set of patients is available in MIMIC II. There are approximately 10,000 patients whom have associated ABP waveforms in multiple contiguous segments spanning minutes to hours to even days which were recorded during their stay in the ICU. Before scaling to the entire dataset and learning with many more features, we have selected a subset of 42 patients who have previously been selected as part of a competition to predict "acute hypotensive events"[2]. The feature conditioned waveforms comprise the dataset of the experiments reported in this publication.

2.1 ABP signal

Arterial blood pressure signal is a periodic signal whose period strongly correlates with the frequency of a heart beat. The beat has three micro structures. The first corresponds to the rise of the pressure from a minimum value (*diastole*) to a peak which is called a *systole*, the second corresponds to the period from the peak to the end of the *systole*, subsequently the pressure rises by a small fraction before returning to a *diastole*. Pressure values are in mmHg and values at the points mentioned above are called *systolic* pressure, P_s, and *diastolic* pressure, P_d, respectively. The time difference between two consecutive *diastoles* is the duration of the beat T_d. A mean arterial pressure (MAP) often used in medicine as measure of normality in blood pressure is defined as $\frac{2.P_d+P_s}{3}$. The values of P_s, P_d and T_d are different from patient to patient and also change over time for a single patient. In addition, an important aspect of this data, when compared to traditional periodic signals is the changes in the beat duration.

2.2 Problem definition

We now define in detail a prediction problem for the machine learning community. As illustrated in Figure 1, we assume a memory(lag), m relative to the current time point, T_0. The aim is to predict MAP for a forecast period, p, for some forecast window, f in the future. In the experiments of this paper, we try to predict the MAP for the time $[T_0 + f, T_0 + f + p]$. Because a precise continuous value of MAP has minimal use in an ICU setting, we discretize the MAP into intervals. This follows clinician intuition to somewhat comfortably employ thresholds and categorize a precise value as *high*, *normal*, and *low*. With this in mind, and after discussions with doctors, we transform the continuous prediction problem into a 3-label classification problem by setting the following MAP thresholds for different intervals

[2]http://www.physionet.org/challenge/2009/

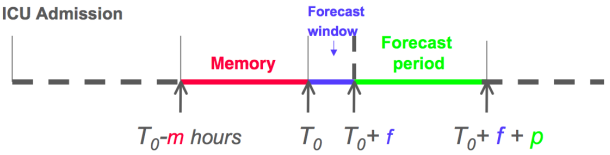


Figure 1: Visual depiction of the ABP predicition problem.

Variable	Name	Domain
V1	Systolic pressure P_s	Time
V2	Beat duration	Time
V3	Standard deviation of signal	Time
V4	Mean of signal	Time
V5	RMS of power spectrum	Frequency

Table 1: List of features extracted per beat.

and assigning each interval a label:

$$\begin{aligned} \text{Low}(0) &\leq 55\text{mmHg} \\ 55\text{mmHg} &< \text{Normal}(1) \leq 85\text{mmHg} \\ 85\text{mmHg} &< \text{High}(2) \end{aligned}$$

The problem definition is parameterized by variables m, f and p. In this paper we use a lag of $m = 100$ interpolated beats, forecast (lead) time window $f = 30$ minutes and a forecast period $p = 10$ minutes.

2.3 Assembling the Training Set

To compile our learning set, we start by marking the onset of each beat from the raw waveform signal sampled at 125Hz using a beat onset detection algorithm available from the MIMIC II database [27]. For the samples that correspond to a beat, we extract 2 features specific to blood pressure: beat duration and systolic pressure. In the time domain we extract the mean and standard deviation of the beat's signal as the 3rd and 4th features. For the 5th and final feature, we consider the spectral frequency and extract the root mean square (RMS) of the power spectrum. These features are listed and given variable names in Table 1.

One minute may comprise 60 beats for a patient at one time and 70 at another. Because of this difference in duration, careful attention is required to establish uniformity throughout the training set. We employ an algorithm by Mori et al. [16] which interpolates the beats to obtain regularly sampled values plus extracts and labels features.

To format the data for EC-Star, which requires data packages, we divide the signal for each patient into data packages with 1500 rows, i.e. 1500 training cases. Each row is an interpolated sample and each sample has a number of features. The columns represent the features. We have 692 data packages, or fitness/training cases, in the training set and 298 data packages as out-of-sample testing set.

A brief descriptive analysis of the data shows that the classes are not clearly separated when each variable for each class is inspected given the lead time $f = 30$min. In addition, the occurrence of classes also shows that the data is unbalanced, there are roughly 1e6 low, 8.5e6 normal and 3.7e6 high labels. Moreover, transitions show that patients

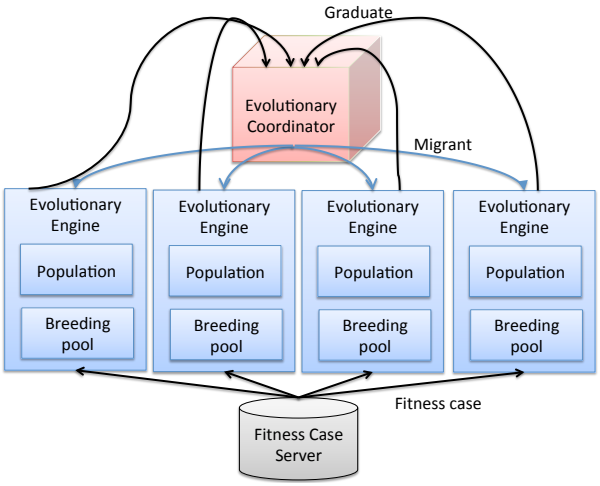


Figure 2: The hub and spoke model of EC-Star.

only rarely go from normal to low and normal to high, and vice versa.

3. EC-Star

EC-Star is a distributed EC system, see Fig. 3, that employs a *modified* decision list representation [19]. It uses a large number of volunteer compute nodes as "Evolutionary Engines", described in detail in Sect. 3.2. The individual solutions from the Evolutionary Engines are coordinated by an Evolutionary Coordinator (see Sect. 3.1), and data is distributed to multiple Evolutionary Engines by a data server.

Most relevant to this contribution's exploration into efficient use of training data is the fact that EC-Star is *distributed*. At full scale, it uses a large number (eventually hundreds of thousands) of volunteer compute nodes. Each Evolutionary Engine is a local LCS (with some variations vs a typical LCS). Like all LCS at a high level, it executes evolutionary algorithm (EA) which iterates over a population of candidate classifier rulesets with selection and reproduction with variation. It only executes during the idle (a.k.a "spare") cycles of its volunteer node and it must accommodate its "host" in a number of ways. In this contribution we address how to resolve the following challenges that are imposed by volunteer compute node requirements in combination with a large training set:

- A Evolutionary Engine can't keep all the training data on the RAM or disk of its host.
- The data server can't track which training cases it has given to an Evolutionary Engine because there could be hundreds of thousands of them eventually. Therefore it **must** dispatch random training cases.
- Tracking training cases is (also) too costly in computation and memory of a Evolutionary Engine.

- It is infeasible to append a long list of training cases to a candidate ruleset because of memory constraints at the Evolutionary Engine and because rulesets get sent around the network on limited capacity channels.

Finally, EC-Star's functions even as Evolutionary Engines volunteer and retire, since the Evolutionary Coordinator has an archive of data (the *migrants*) from any Evolutionary Engine that was active long enough. In addition, the data-server only sends training cases in response to a request from a Evolutionary Engine.

3.1 Evolutionary Coordinator

The Evolutionary Coordinator coordinates migration among the Evolutionary Engines and maintains a layered *archive* – a sorted set of individuals that are currently the best from all Evolutionary Engines, see Alg. 1. The archive allows individuals to be sent out to each Evolutionary Engine in order to improve fitness and possibly mix the existing genetic material. When a *migrant* message is received the returning individuals are competing for the space in the archive. `archive_migrants()` is called and individuals compete for a slot if the archive is already full. When an `migrant_query` message is received *migrants* are randomly picked from the Evolutionary Coordinator and sent to the requesting Evolutionary Engine.

Algorithm 1 Evolutionary Coordinator

```
1: initialize()
2: loop
3:     message = listen()
4:     if message == migrants then
5:         archive_migrants(migrants)
6:     else if message == migrant_query then
7:         migrants = get_random_migrants(archive)
8:         send_migrants(migrants)
```

3.2 Evolutionary Engine

Each Evolutionary Engine runs a completely independent evolutionary algorithm, pseudocode is shown in Alg. 2. It has a fixed `pop_size` and initially generates the population randomly. In the evolutionary loop, it requests training cases (training data) from the training case server in the form of a *data package* . Each individual in the population is evaluated, and after a fixed number of data packages– `evals_per_generation`– selection and breeding take place before replacement and the next generation, see Sec. 3, Alg. 3 and Alg. 4 for further details. Periodically local individuals become graduates and are dispatched to the Evolutionary Coordinator and *migrants* are received from the Evolutionary Coordinator.

Algorithm 2 Evolutionary Engine: Evolution

```
1: population = initialize()
2: loop
3:     for all i ∈ evals_per_generation do
4:         sample = get_rnd_data_package()
5:         for all ind ∈ population do
6:             ind.num_evals + +
7:             for all event ∈ sample do
8:                 prediction = sample_prediction(sample, ind)
9:                 saved_predictions.append(prediction)
10:            ind.fitness = calculate_fitness(saved_predictions)
11:            ind.absolute_fitness = ind.fitness
12:            ind.relative_fitness = ind.absolute_fitness/ind.num_evals
13:    report_to_server(migrants)
14:    breeding_pool = select_breeders(population)
15:    graduates = select_graduates(breeding_pool)
16:    report_to_server(graduates)
17:    population = create_next_generation(breeding_pool)
18:    breeding_pool = []
```

3.3 Representation

The representation of individuals in EC-Star is similar to a classifier in the so-called *Pittsburgh-Style* version of a LCS [12]. As in a Pittsburgh-Style LCS, the rule set is assigned a fitness and the individual represents a full solution-space – that is, each individual contains the rules needed to classify a row of test data.

An individual (*classifier*) is a header with id, age and fitness and a body with a set of rules. Each rule is a variable length conjunctive set of conditions with an associated action which is a class in a classification problem. Each condition acts as a propositional variable, which is then applied to the discretized real-valued training environment. Apart from conjunction operators each condition can have, a *complement* operator, negating the truth value and a *lag* which refers to "past" values of an attribute, in the EC-Star representation this is previous rows in the training case (data package). The lag allows the rule to consider a variables value in previous cases when testing a current case. See below for an example of an individual.

```
!(V3[7] < -25.0) & !(V3[3] < -40.0) --> Label == 1
V3[5] < 20.0 & !(V2[4] < -14.0) --> Label == 2
V5[7] < 294000000.0 --> Label == 2
V1[21] < 42 --> Label == 0
```

3.4 Training Set Evaluation

Each individual in an Evolutionary Engine is evaluated on a number of training cases every generation. Each training case is in a data package and consists of a number of rows, called events. For each event, the variables in the classifier's rules' conditions are bound to the features of the event. For each rule in the individual, the rule's conditions are evaluated. If all are true, the rule is added to a *active set*. Finally, a voting mechanism elects from the *active set* a single rule's action as a prediction for the training case. When there are no active rules no prediction is made. We track *activity*, the number of times each rule is active, and use is as a basis for selection into the breeding pool, i.e. an individual must have had some active rules. Predictions for all events in a training case are verified against the true class labels. A correct prediction increases the fitness and incorrect decreases. Each individual records its fitness in two ways: `relative_fitness` and `absolute_fitness`. See Algorithm 2, lines 7-12 for pseudocode of this process.

3.5 Related Work

While previous work has studied training set use or LCS design, none have combined sub sampling fitness with bracketing and early stopping with a large scale LCS executing on volunteer compute nodes.

The bracketing strategy in EC-Star ensures that only individuals which have been evaluated on the same number of training cases compete. It is important not to confuse this with age layering as in ALPS, [8]. Age in ALPS denotes how many generations an individual descends from. The early stopping in [5] is the same as our strategy in the sense that both terminate fitness testing if a solution is not good enough. Jin [10] has a survey about fitness approximation by subsampling. In [11] subsampling helps resolves surrogate model inaccuracy.

A review of efficiency enhancement mechanisms for GBML methods for large-scale data mining using GBML is available in [3]. Examples comprise: windowing mechanisms, hybrid methods, fitness surrogates, hardware utilization, ensemble mechanisms and parallel models. EC-Star can be considered a windowing scheme, where `evals_per_generation` age is the window and the samples in the window are uniformly randomly drawn from the training set.

Bacardit et al. [2] speeds up the modeling time and accuracy of an LCS by stratifying the training set into subsets. Each strata maintains approximately the class distribution of the whole training set and a round-robin policy selects different strata for the GA iteration's fitness computation. Ishibuchi et al. [9] divides the training data and rotates it with a fuzzy hybrid genetics-based machine learning system which uses LCS.

Other ad-hoc parallel architectures are: GALE which has a lattice-based cellular GA architecture, with different cells in the lattice having different subsets of training examples assigned to them [13] and the NAX system which assigns different subsets of the training data to different nodes, while the GA cycle is run redundantly since the population is replicated on all nodes only synchronizing the fitness evaluations between the nodes [14]. Scheidler and Middendorf [20] solve classification problems in computing systems that consist of distributed, memory constrained components. Interacting Pittsburgh-style LCSs are used to generate sets of classification rules that can be deployed on the components and enables the components to solve complex classification problems in cooperation.

Recent work in improving LCS performance when using rulesets includes [7] who post-process decision lists, [1] who consider feature selection and [21] who use expert knowledge to guide search more efficiently. Urbanowicz et al. [22] introduces random artificial noise in a learning classifier system environment and slightly improves its testing accuracy.

There are other LCS systems solving classification in medical data, like EC-Star. Bojarczuk et al. [4] uses a constrained syntax GP system with a hybrid Pitt/Michigan approach that discovers classification rules in medical data sets. An ML framework for medical classification that is scaled for grid computing is described in Ramos-Pollán et al. [18].

Scalability has been addressed in a number of different ways. Some, like EC-Star, distribute the algorithm. For example, Urbanowicz et al. [23] use a cluster and Franco et al. [6] use a GPU. EC-Star's use of volunteer compute nodes appears to be unique and allows many more resources (in the hundreds or thousands) to be enlisted. Fernández de Vega et al. [25] go through customizable execution environments for evolutionary computation using BOINC+ virtualization. Merelo et al. [15] discuss pool vs. island based evolutionary algorithms. IslandSofEA uses separate clients and a pool, this scales best of their tested methods. This is similar setup to EC-Star.

4. STRATEGIC USE OF TRAINING DATA

We are concerned that executing even one classifier ruleset against all the training data will be extremely costly. Our strategy has a number of steps: Locally at each evolutionary engine, we first cull individuals that perform relatively worse on the basis of a small fraction of the training set, while allowing those that are relatively better to survive. These survivors are next "seasoned" locally on more training cases. During seasoning, they are used for breeding. At the end of seasoning, if they pass a minimum fitness standard, they become "graduates" of their evolutionary engine and

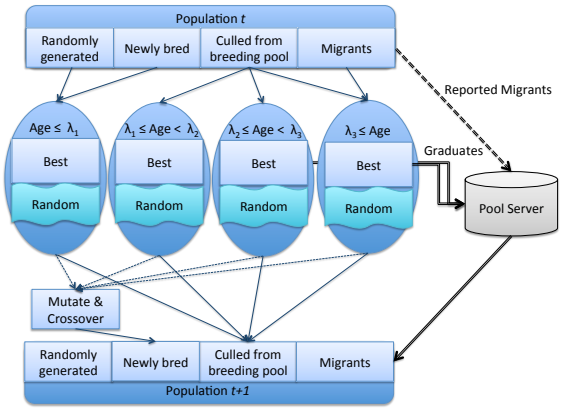


Figure 3: Evolution at the evolutionary engine.

are passed to the "Evolutionary Coordinator". It archives the best graduates it receives and randomly sends them out to evolutionary engines as *migrants* for further evaluation.

Migrants are returned periodically by evolutionary engines but how long it will take for this return is unpredictable. The Evolutionary Coordinator has to strategically deal with this evaluation quantity mismatch while it exerts selection pressure on *migrants* to keep their quantity in check. To do this, its archive is a set of fixed size brackets each spanning successively higher intervals of fitness evaluations. A truly fit individual moves up the brackets as it is evaluated on more training cases. Eventually it reaches the topmost bracket and is harvested from the system as a high performing solution.

4.1 Logic at the evolutionary engine

At the evolutionary engine we use `evals_per_generation` as the number of training cases evaluated per generation. `min_evals_graduation` is the minimum number of training cases a individual must have been evaluated on, before graduation. `pop_size` is population size and `breeding_pool_size` is $0.2 * \texttt{pop\_size}$.

The evolutionary engine admits selected population members to the `breeding_pool` and creates the next generation per lines 17 and 20 in Alg. 2. Alg. 3 provides pseudocode of its `breeding_pool` selection logic and Alg. 4 explains how the next generation is generated, also see Figure 3.

The general strategy is to evaluate all completely unevaluated local individuals on just a small fraction, $\lambda_1 =$ `evals_per_generation`, of the training set. Comparisons between locals incur low coverage but is completely symmetric. Among these new individuals, only a small superior proportion will be propagated directly to the next generation and also be copied to the breeding pool. The rest are discarded. We decide not to waste effort evaluating them further. To gain admission to the `breeding_pool`, a new individual's `relative_fitness` must be better than that of the weakest new individual already in the breeding pool and it must be sufficiently different from others already in the pool in terms of `relative_fitness` and activity, see `filter_1` and `qualify_to_enter()` in Algorithm 3.

The superior individuals are then allowed to "season" for

Algorithm 3 Breeding

```
1: function select_breeders(population)
2:     for all ind ∈ population do
3:         if ind.num_evals < λ1
   &&qualify_to_enter_bracket(ind, brackets.lowest) then
4:             brackets.lowest.insert(ind)
5:         else if ind.num_eval < λ2 then
6:             brackets.seasoning.append(ind)
7:         else if ind.num_eval < λ3 then
8:             brackets.potential_graduation.append(ind)
9:         else if qualify_to_enter_bracket(ind, bracket.highest) then
10:            brackets.highest.insert(ind)
11:    for i ∈ DiversityQuota do
12:        ind = select_randomly_one_ind(population)
13:        bracket = find_bracket(ind.num_evals)
14:        if filter_fitness(ind, bracket) then
15:            bracket.append(ind)
16: function filter_fitness(ind, bracket)
17:    return dist(ind.relative_fitness, bracket) > k1
18: function qualify_to_enter_bracket(ind, bracket)
19:    filter2 = TRUE
20:    if bracket.quantity == bracket.size then
21:        worst = min_relative_fitness(bracket)
22:        filter2 = worst.relative_fitness < ind.relative_fitness
23:    return filter_fitness(ind, bracket)&&filter2
```

Algorithm 4 Next Generation

```
1: function create_next_generation(breeding_pool)
2:     evolved_new_inds = breed(breeding_pool)
3:     guests = get_new_guests_from_server()
4:     culled_breeding_pool = breeding_pool.remove(graduates)
5:     culled_breeding_pool = culled_breeding_pool.remove(migrants)
6:     population.append(evolved_new_inds, migrants, culled_breeding_pool)
7:     while population.quantity < population.size do
8:         population.append(make_random_ind)
9: function select_graduates(breeding_pool)
10:    graduates1 = filter_brackets.potential_graduation(activity > C)
11:    graduates2 = filter_brackets.highest(activivty > C&&!ind.migrant
12:    return graduates1 ∪ graduates2
```

a few more generations until they have λ_2 fitness evaluations. Seasoning implies that, without competition, every generation they are both copied to the next generation and the breeding pool. After λ_2 fitness evaluations, again they are copied to the breeding pool. However, all with number of evaluations $< \lambda_3$ who have been active enough graduate to the Evolutionary Coordinator rather than get copied to the next generation. After λ_3 fitness evaluations, those who have not graduated compete with *migrants* (who also exceed λ_3 fitness evaluations) to graduate.

Figure 3 shows the thresholds λ_1, λ_2, and λ_3, on the breeding pool. These thresholds form 4 evaluation brackets, which we call `lowest`, `seasoning`, `potential_graduation` and `highest`.

The population also contains *migrants*. Each generation the *migrants* are evaluated with the same training cases as individuals locally created on the evolutionary engine. They are always reported to the Evolutionary Coordinator. They compete for entry into the breeding pool's `highest` bracket with local individuals with more than λ_3 fitness evaluations. The next generation they are replaced with new *migrants* from the Evolutionary Coordinator.

After every individual in the population has been tested for admission to the `breeding_pool`, it is supplemented with random individuals from the population for diversity. Random individuals enter a bracket only if they are different from existing individuals in the bracket. Then the `breeding_pool` is used for reproduction. Parents are selected from it randomly and offspring are created through copying, crossover and mutation.

The next generation is partly composed of the breeding pool which is culled of *migrants* and graduates. It also includes new *migrants*. Added to it are evolved offspring – their breeding continues until, accounting only 10% of the pool remains. This remaining fraction is filled out with random individuals.

4.2 Logic at the Evolutionary Coordinator

Our strategy at the Evolutionary Coordinator employs evaluation brackets to isolate competition in the Evolutionary Coordinator to occur between individuals of approximately the same coverage. Each bracket spans a successive range of fitness evaluations and typically the range is much bigger than the evaluations necessary to graduate from the evolutionary engine (`min_evals_graduation`). When a new graduate goes to the Evolutionary Coordinator, it competes for entry at the lowest bracket. Each time a *migrant* is reported by a evolutionary engine, its number of fitness evaluations has increased by `evals_per_generation` and, to re-enter the Evolutionary Coordinator, it enters in the appropriate evaluation bracket if `relative_fitness` is better than the worst individuals.

At lower brackets, when the range of a bracket is small relative to the size of the fitness suite, coverage is lower and symmetry is poor, so competition in these brackets often results in what might be considered an error: one individual is determined to be better than another based on what training cases each have been evaluated on, but, on the entire fitness suite it is actually inferior. At higher layers, where coverage may exceed the size of the fitness suite (to compensate for the effects of randomized sampling of the fitness suite), symmetry improves and competition is less error-prone.

5. EXPERIMENTS

Our first experimental question is how the evolutionary engine parameter, `evals_per_generation`, which controls the quantity of partial fitness evaluations, influences predictive accuracy on the ABP test set. From this evaluation we choose the best of the three parameter settings we tried and use it to solve the ABP prediction problem described in Section 2. We report the test set accuracy of EC-Star and a neural network.

5.1 Effect of `evals_per_generation` Parameter

The `evals_per_generation` parameter is important because it determines how many training cases individuals are evaluated against, at the evolutionary engine, before the first culling step. If this quantity is too low, the system would be vulnerable to making poor early choices because fitness estimates at this step are too unreliable. If the quantity is too high, fitness evaluations are wasted on individuals that are "already" fit enough to breed, migrate and provide a good solution. We therefore focus on whether there is a range of values for this parameter that allow the strategy to overcome the conditions imposed by using volunteer compute nodes.

We vary `evals_per_generation` with settings equal to $2, 10, 100$. Each run lasts for 2.5 hours using 10 clients for each `evals_per_generation` setting and one evaluation of a training case takes about 3 seconds. The limit in the archive on the Evolutionary Coordinator was set to be very high, roughly 7X the size of the training set, and each bracket was a 50th of the set size.

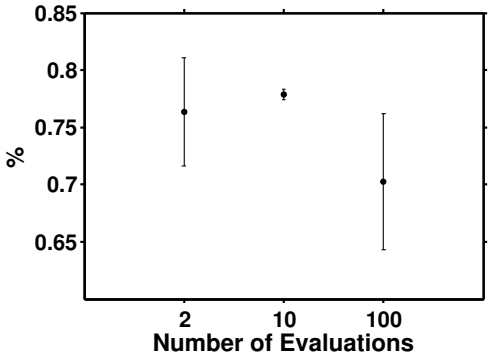


Figure 4: Test accuracy average for 12 runs for each setup after 2.5h for the best of the top 10 individuals in the Evolutionary Coordinator

We show test results in Figure 4. We might expect performance to be more varied when a low `evals_per_generation` is used since fewer training cases are evaluated for each generation because, when the number of training cases evaluated at a generation is low, the comparisons becomes less symmetric. We, in fact, observe variance among the runs for settings 2 and 100 but not for 10. The reasons why 10 has very narrow variance is not apparent to us. In terms of accuracy, a t-test with $\alpha = 0.05$ shows that there is significant difference in accuracy between `evals_per_generation` settings 100-2 and 100-10. There is no significant difference between `evals_per_generation` 10-2.

For the experiments in Section 5.2 we then selected a `evals_per_generation` of 10 because it has the highest mean test accuracy and lowest variance over the 12 runs.

We also tested setting `evals_per_generation` to the maximum number of data packages used for training (692) so that each solution on the Evolutionary Engines is always evaluated on all the training cases. However, only 2 runs produced results within the assigned time and they had a lower test accuracy of approximately 70%.

5.2 Blood pressure performance Results

Setting `evals_per_generation`, i.e. λ_1 10, $\lambda_2 = 4 * \lambda_1$ and $\lambda_3 = 5 * \lambda_1$, we next generate performance results. The Evolutionary Coordinator bracket range is 20 and max number of evaluations is 1000. The data is randomly split into test and training, 70% of the data is used for training and the rest is used for out-of-sample testing. This procedure is repeated 10 times in order to create 10 different training and test data sets. From each of 6 separate runs of 1h with 13 Evolutionary Engines with a population size of 500 we take the individual with the best relative fitness at the Evolutionary Coordinator. The average median test accuracy on all the splits is 77.95%±2.6%.

We perform an approximate comparison of our prediction accuracy versus that of a neural network from the neural network toolbox in MATLAB[3]. Before running the neural network we were able to remove noise from the experimental dataset. We used the same 5 features but did not add more to allow lags to be referenced. In comparison, EC-Star used 500 features when considering the lag of 100 rows in the data package. Per toolbox input structure, we performed 10-fold

[3]`http://www.mathworks.com/products/neural-network/`

cross validation with 50 runs on each fold (for this contribution do not have time to re run EC-Star). Each neural network's training time was approximately 5 min and the median accuracy was approximately 80%, which is approximately as good as EC-Star's. Direct comparison, because of the slightly different data sets, is not possible but, very roughly, the results are in the same realm. The search space the neural network learner had to navigate was smaller while the solutions from EC-Star are rules which can be easier to interpret for humans. It is important to stress that we consider these results preliminary because the data set we are currently using is small and very unbalanced.

5.3 Analysing the Training Set Use

Two aspects of the training set in EC-Star are described. First, EC-Star does not guarantee that an individual ha covered all the training cases. Although, when more training cases are shown the probability of training on all increases. In addition, when two individiuals are compared there is no guarantee that they have trained on the same cases, i.e. the symmetery is different. Again, if the number of training cases that an individual has been exposed to increases, then the probability of a symmetric comparison increases.

We first consider the expected coverage on the training cases, i.e. number of fitness evaluations needed for an individual to be evaluated on every case in the fitness suite. The coverage can be studied in regards to the max evaluations at the Evolutionary Coordinator. E.g. at what max evaluations is and individual expected to have been evaluated on all data packages. The probability for a solution to pick all the data packages is a version of the coupon collector problem Motwani and Raghavan [17]. We ask how many coupons (data packages) must be drawn with replacement before all the coupons have been collected. E.g. $N = 692$ coupons, $E[t = N]$ is the expected number of draws to collect N coupons, $\gamma \approx 0.5772156649$.

$$\begin{aligned} E[t=N] =& 1 + \frac{N}{N-1} + \frac{N}{N-2} + \cdots + N \\ =& NlnN + \gamma N + 1/2 + O(1/N) \\ E[t=692] =& 692ln(692) + \gamma N + 1/2 + O(1/N) \\ \approx& 4862 \end{aligned}$$

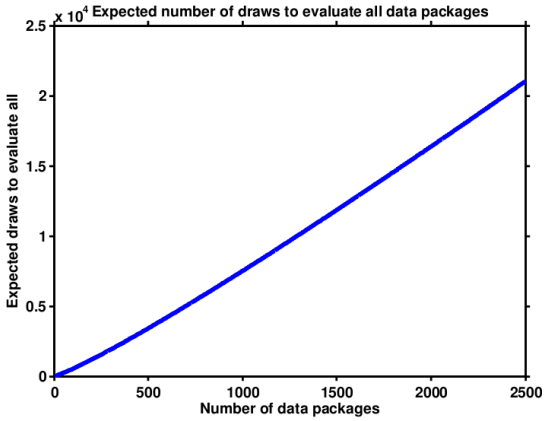
Figure 5(a) plots the expected number of fitness evaluations given different numbers of data packages. Our fitness suite size of 692 gives 4862 as the number of packages needed to be evaluated before all 692 have been evaluated.

We also considered symmetry given training case randomization, it is possible to calculate the probability that two individuals see the same data package. i.e. the symmetry of individuals' training cases. We rely upon a variant of the generalized birthday problem [26] which derives the likelihood of two people's birthdays, m and n, in a group of size d occuring on the same day (i.e. a "collision"). We substitute the (num_evals) number of training cases two individuals have been evaluated upon for the birthdays and the fitness suite size for d and a "collision" is a common training case. The probability of a collision can be approximated by:

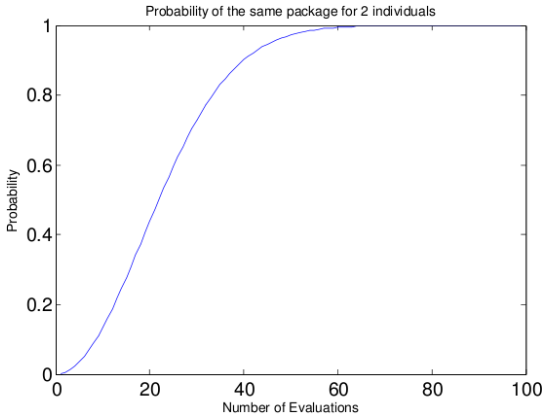
$$p(m, n, d) = 1 - (1 - 1/d)^{mn}$$

The probability is plotted in Figure 5(b) with $d = 692$ and number of evaluations, $m = n$.

Symmetry is controlled by altering `evals_per_generation`



(a) data package coverage



(b) Same data package (total 692)

Figure 5: Coverage and symmetry approximations.

and in the lowest bracket of the Evolutionary Coordinator, the likelihood of a common training case is lowest. As individuals flow up the evaluation brackets, i.e. coverage increases, on the Evolutionary Coordinator. There are greater odds that there will be some overlap in the training cases which were evaluated by each individual gains higher coverage. The design of EC-Star allows the symmetry and coverage to increase as an individual is exposed to training cases.

6. CONCLUSIONS & FUTURE WORK

In this paper, we presented a new definition of a blood pressure prediction problem involving 3 label classification. Because the data for learning a LCS ruleset for this purpose will eventually be very large in number of time segment and wide in terms of features, we used a modest dataset and started to examine how we could circumvent the very significant expense of fitness evaluation which would occur in the future when we use EC-Star with very large training data. We exploited comparing individuals early in their"life" on the basis of a smaller fraction of the training set. Only survivors are further tested and competition compares only individuals of the same training set exposure through an archive layering mechanism. We examined the impact of our strategy on learning performance. The approach avoids the overhead of predetermining the division of training cases and the question of how we would split them deterministi-

cally. It has a more modest demand on bandwidth between the training case server and learners. It avoids centralized management of sampling which would be needed if it was not random avoids precise retrieval by the data server too.

For future work one improvement of the design would be to have different selection pressure at the different layers, e.g. using tournament or roulette wheel selection. We could balance the data and fitness function costs. In addition, we expect more challenge with a larger fitness suite (from more patients) though we may not see any significant change in (im)balance. This indicates we should try weighting the cost of different types of errors. We also intend to investigate different thresholds for the features. For the EC-Star architectur we will investigate how the unreliability of the volunteer compute node network affects the results. Moreover, different distributions of data packages can also be tested.

Acknowledgements

We thank the Li Ka Shing Foundation and Genetic Finance.

References

[1] Bacardit, J., Burke, E., Krasnogor, N.: Improving the scalability of rule-based evolutionary learning. Memetic Computing 1(1), 55–67 (2009)

[2] Bacardit, J., Goldberg, D.E., Butz, M.V., Llorà, X., Garrell, J.M.: Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy. PPSN VIII. pp. 1021–1031. Springer (2004)

[3] Bacardit, J., Llorà, X.: Large-scale data mining using genetics-based machine learning. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 3(1), 37–61 (2013)

[4] Bojarczuk, C., Lopes, H., Freitas, A., Michalkiewicz, E.: A constrained-syntax genetic programming system for discovering classification rules: application to medical data sets. Artificial Intelligence in Medicine 30(1), 27–48 (2004)

[5] Bongard, J., Hornby, G.: Guarding against premature convergence while accelerating evolutionary search. GECCO, pp. 111–118. ACM (2010)

[6] Franco, M., Krasnogor, N., Bacardit, J.: Speeding up the evaluation of evolutionary learning systems using gpgpus. GECCO, pp. 1039–1046. ACM (2010)

[7] Franco, M.A., Krasnogor, N., Bacardit, J.: Post-processing operators for decision lists. GECCO, pp. 847–854. GECCO, (2012),

[8] Hornby, G.: Alps: the age-layered population structure for reducing the problem of premature convergence. GECCO, pp. 815–822. ACM (2006)

[9] Ishibuchi, H., Mihara, S., Nojima, Y.: Training data subdivision and periodical rotation in hybrid fuzzy genetics-based machine learning. In: Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on. vol. 1, pp. 229–234. IEEE (2011)

[10] Jin, Y.: A comprehensive survey of fitness approximation in evolutionary computation. Soft Computing-A Fusion of Foundations, Methodologies and Applications 9(1), 3–12 (2005)

[11] Jin, Y.: Surrogate-assisted evolutionary computation: Recent advances and future challenges. Swarm and Evolutionary Computation (1), 61–70 (2011)

[12] Jong, K.A.D., Spears, W.M., Gordon, D.F.: Using genetic algorithms for concept learning. Machine Learning 13, 161 (1993)

[13] Llora, X., Garrell, J.M., et al.: Knowledge-independent data mining with fine-grained parallel evolutionary algorithms. GECCO, pp. 461–468. Citeseer (2001)

[14] Llorà, X., Reddy, R., Matesic, B., Bhargava, R.: Towards better than human capability in diagnosing prostate cancer using infrared spectroscopic imaging. GECCO pp. 2098–2105. ACM (2007)

[15] Merelo, J., Mora, A., Fernandes, C., Esparcia-Alcazar, A.I., Laredo, J.L.: Pool vs. island based evolutionary algorithms: an initial exploration. In: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 19–24. IEEE (2012)

[16] Mori, N., Suzuki, T., Kakuno, S.: Noise of acoustic doppler velocimeter data in bubbly flows. Journal of engineering mechanics 133(1), 122–125 (2007)

[17] Motwani, R., Raghavan, P.: Randomized algorithms. Cambridge university press (1995)

[18] Ramos-Pollán, R., Guevara-López, M., Oliveira, E.: A software framework for building biomedical machine learning classifiers through grid computing resources. Journal of medical systems pp. 1–13 (2012)

[19] Rivest, R.: Learning decision lists. Machine learning 2(3), 229–246 (1987)

[20] Scheidler, A., Middendorf, M.: Learning classifier systems to evolve classification rules for systems of memory constrained components. Evolutionary Intelligence 4(3), 127–143 (2011)

[21] Urbanowicz, R., Granizo-Mackenzie, D., Moore, J.: Using expert knowledge to guide covering and mutation in a michigan style learning classifier system to detect epistasis and heterogeneity. PPSN XII pp. 266–275 (2012)

[22] Urbanowicz, R., Sinnott-Armstrong, N., Moore, J.: Random artificial incorporation of noise in a learning classifier system environment. GECCO, pp. 369–374. ACM (2011)

[23] Urbanowicz, R., Granizo-Mackenzie, A., Moore, J.: An analysis pipeline with statistical and visualization-guided knowledge discovery for michigan-style learning classifier systems. Computational Intelligence Magazine, IEEE 7(4), 35–45 (2012)

[24] Urbanowicz, R., Moore, J.: Learning classifier systems: a complete introduction, review, and roadmap. Journal of Artificial Evolution and Applications 2009, 1 (2009)

[25] Fernández de Vega, F., Olague, G., Trujillo, L., Lombraña González, D.: Customizable execution environments for evolutionary computation using boinc+ virtualization. Natural Computing pp. 1–15 (2012)

[26] Wendl, M.C.: Collision probability between sets of random variables. Statistics & Probability Letters 64(3), 249 – 254 (2003)

[27] Zong, W., Heldt, T., Moody, G., Mark, R.: An open-source algorithm to detect onset of arterial blood pressure pulses. In: Computers in Cardiology, 2003. pp. 259–262. IEEE (2003)