Collaborative Expert Portfolio Management

David Stern
Ralf Herbrich
Thore Graepel
Microsoft Research Ltd.
Cambridge, UK
{dstern,rherb,thoreg}@microsoft.com

Horst Samulowitz
National ICT Australia
and University of Melbourne,
Victoria, Australia
horst@csse.unimelb.edu.au

Luca Pulina
Armando Tacchella
Universita di Genova
Viale Causa,
Genova, Italy
{luca.pulina,tac}@dist.unige.it

Abstract

We consider the task of assigning experts from a portfolio of specialists in order to solve a set of tasks. We apply a Bayesian model which combines collaborative filtering with a feature-based description of tasks and experts to yield a general framework for managing a portfolio of experts. The model learns an embedding of tasks and problems into a latent space in which affinity is measured by the inner product. The model can be trained incrementally and can track non-stationary data, tracking potentially changing expert and task characteristics. The approach allows us to use a principled decision theoretic framework for expert selection, allowing the user to choose a utility function that best suits their objectives. The model component for taking into account the performance feedback data is pluggable, allowing flexibility. We apply the model to manage a portfolio of algorithms to solve hard combinatorial problems. This is a well studied area and we demonstrate a large improvement on the state of the art in one domain (constraint solving) and in a second domain (combinatorial auctions) created a portfolio that performed significantly better than any single algorithm.

Introduction

When faced with a diverse set of difficult tasks, a general approach which would be effective for all of them is likely to be complex. In a given domain a specialist approach to a subset of tasks is often simpler and more effective at those tasks. This suggests a divide and conquer approach using a portfolio of specialist experts.

An example of this is the case where the expert is an algorithm [Smith-Miles, 2008; Gomes and Selman, 1997; Horvitz et al., 2001; Rice, 1976; Streeter and Smith, 2008]. A portfolio approach means that several parallel and independent efforts to design an algorithm for a task can be combined so as to leverage the best aspects of each of them. An example class of tasks where an algorithm portfolio can be beneficial is hard combinatorial problems where typically there does not exist one single approach that outperforms any other across a range of real-world problems [Xu et al., 2008; Leyton-Brown, Nudelman, and Shoham, 2009]. Because of their exponentially large search spaces, these problems are in general computationally intractable and depending on the properties of the problem one particular solver performs better than another according to some measure (e.g., run-time). A vast discrepancy in performance arises because solvers commit to different heuristic choices or employ different techniques. There is a successful line of research showing that the variance in algorithm performance can be exploited by machine learning methods to produce a portfolio of algorithms that greatly outperforms any individual algorithm in domains such as constraint reasoning [Pulina and Tacchella, 2009; Xu et al., 2008; Gomes and Selman, 1997] and combinatorial auctions [Leyton-Brown, Nudelman, and Shoham, 2009].

The goal of this paper is to address some of the limitations of previous work and provide a general framework for solving diverse tasks with a portfolio of experts. Besides algorithms, other examples of expert portfolios to which we are interested in applying this approach include programming languages[1] and Yahoo Answers (where the experts are humans). We consider the following properties desirable:

- The system should select a specific scheduling strategy for each task (based on task features) [Streeter and Smith, 2008].
- The model must be trained on-line so the model can immediately take account of each outcome to improve future decisions. The computation cost of this update should not depend on the number of previously seen problems [Pulina and Tacchella, 2008].
- The system should adapt continuously over time, tracking a changing domain and changing expert characteristics.
- There should be a principled method to choose an expert at any point in time when performing a task.
- The sub-system for taking into account performance should be pluggable as depending on the type of task, feedback comes in different forms. For example, for constraint solving we have data for solution speed whereas for the task of assigning human experts to answer questions we might have human judgments of the quality of the answer.

[1]See `http://shootout.alioth.debian.org/`

We require a model which takes a description of each task (in the form of a set of features) and predicts which expert, from a set of experts, would be best suited to solve it (based on past performance). There appears to be a striking similarity between this problem and the well studied challenge of recommending items to users of a web service (e.g. Amazon or Netflix), where users are analogous to tasks and items are analogous to experts. Two approaches are typically available in that case. Firstly, content-based approaches make use of descriptions (feature vectors) of users and items. For example, items may be described by properties such as author and manufacturer. Secondly, collaborative filtering approaches allow us to learn about a user by the items they have previously rated and the users who have rated items in common with them, using implicit descriptions of users and items obtained from a (sparse) matrix of previous ratings of items by users [Breese, Heckerman, and Kadie, 1998; Varian and Resnick, 1997].

Matchbox [Stern, Herbrich, and Graepel, 2009] is a recommendation system model which combines these two sources of information and here we apply it to recommending experts to tasks. Unlike other collaborative filtering models, Matchbox takes feature vectors of the task and expert as the input. It maps these features into a shared latent space in which the performance of an expert on a task is measured by the inner product. For expert recommendation we crucially rely on this combination of collaborative filtering with generalising features as we usually need to predict the best expert for a task which has not been previously seen. In addition, Matchbox addresses the other desiderata: it is a full Bayesian model and hence allows us to use a principled decision theoretic approach to expert selection, it can be trained incrementally and it is able to dynamically track non-stationary distributions. Finally, it allows free choice of feedback model.

Expert Portfolio Management Model

A Bi-linear Model of Expert Performance Initially, let us assume that each time a task has been attempted by an expert we have available a triple $(\mathbf{x}, \mathbf{y}, r)$ of task descriptions $\mathbf{x} \in \mathbb{R}^n$, expert descriptions $\mathbf{y} \in \mathbb{R}^m$ and a performance score $r \in \mathbb{R}$ indicating the degree of success of the expert on this task. Following Stern, Herbrich, and Graepel [2009], we define the K dimensional task trait vector as $\mathbf{s} = \mathbf{U}\mathbf{x}$ where $\mathbf{U}$ is a $K \times n$ matrix of latent task traits where each element u_{ki} is the contribution of feature i to user trait dimension k. Similarly we define the K dimensional expert trait vector as $\mathbf{t} = \mathbf{V}\mathbf{y}$ for a $K \times m$ expert trait matrix, $\mathbf{V}$. We also model a bias, $b = \mathbf{x}^\top \mathbf{u} + \mathbf{y}^\top \mathbf{v}$ where $\mathbf{u}$ and $\mathbf{v}$ are the biases for each element in the task and expert description vectors. Now, the performance, r, of the expert is modeled as

$$p(r|\mathbf{s}, \mathbf{t}, b) = \mathcal{N}(r|\mathbf{s}^\top \mathbf{t} + b, \beta^2)$$

where β is the standard deviation of the observation noise. Thus we adopt a bi-linear form in which the performance of an expert on a task is given by the inner product of a vector of task traits and a vector of expert traits. The expected rating is proportional to the lengths $||\mathbf{s}||$ and $||\mathbf{t}||$ of the trait vectors

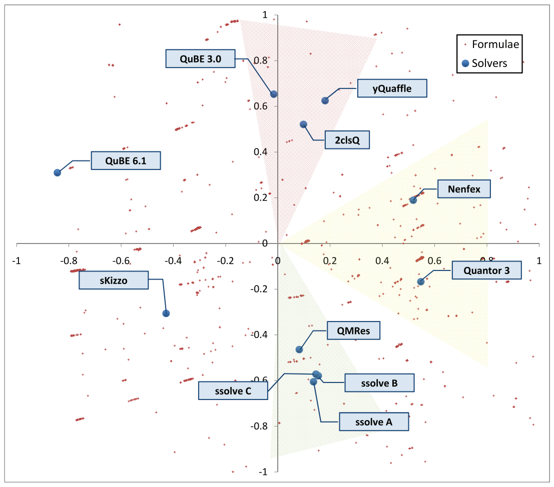

Figure 1: A learned embedding of the 11 solvers and 2570 problems into a 2-dimensional trait space for $K = 2$. The mean of the first solver trait $v_{(0)i}$ is plotted against the mean of the second solver trait $v_{(1)j}$ to produce the blue dots. The mean of the first problem trait $u_{(0)i}$ is plotted against the mean of the second problem trait $u_{(1)j}$ to produce the red crosses. Similarity in this space depends on the angle between trait vectors.

and the cosine of the angle between them.

The model parameters to be learned are the variables $\mathbf{U}$ and $\mathbf{V}$ which determine how tasks and experts are mapped to the K dimensional trait space, and $\mathbf{u}$ and $\mathbf{v}$, the value of bias features for the rating. We represent our prior beliefs about the values of these parameters by independent Gaussian distributions on each of the components of the matrices $\mathbf{U}$ and $\mathbf{V}$ and $\mathbf{u}$ and $\mathbf{v}$. For example we have $p(\mathbf{U}) = \prod_{k=1}^{K} \prod_{i=1}^{n} \mathcal{N}(u_{ki}; \mu_{ki}, \sigma_{ki}^2)$. We choose this factorizing prior because it reduces memory requirements to two parameters (a mean and variance) for each component and allows us to perform efficient inference. We set the mean of each Gaussian prior to zero and the variance to unity.

Or approach allows us to track the performance of experts on tasks dynamically in order to adapt to changing conditions. The characteristics of experts and tasks may change with time. For example if the expert is an algorithm then it could be modified *in situ* by applying an update. We model these dynamics by assuming that the latent variables $\mathbf{U}, \mathbf{V}, \mathbf{u}$ and $\mathbf{v}$ drift with time by the addition of Gaussian noise each time step. For the example we have $p(u_{ki}^{(t+1)}|u_{ki}^{(t)}) = \mathcal{N}(u_{ki}^{(t+1)}; u_{ki}^{(t)}, \gamma^2)$ where t is an index over time steps. At time t_0 we use the prior $p(u_{ki}^{(0)}) = \mathcal{N}(u_{ki}; \mu_{ki}, \sigma_{ki}^2)$. Analogous models may be used for each of the other latent variables.

Inference

The model described above can be further factorized by introducing some intermediate latent variables z_k to represent the result of each component, $s_k t_k$, of the inner product. That is, $p(z_k|s_k, t_k) = \mathbb{I}(z_k = s_k t_k)$. Now the latent perfor-

Solver	# Solved	Mean Time per Task	Utility
2clsQ	738	25.2	-837
Nenofex	866	22.4	-582
QMRes	595	17.5	-1109
Quantor 3.0	834	16.5	-637
QuBE 3.0	1008	13.6	-289
QuBE 6.1	1966	14.3	1603
sKizzo	1723	21.3	1103
ssolve A	837	21.5	-638
ssolve B	840	22.7	-634
ssolve C	818	12.7	-663
yQuaffle	886	13.6	-530

Table 1: Individual solver performances on the 2282 test problems. Shown are the number of solved instances and the mean time per solved instances per solver. The total utility, u, is also shown for $c = \frac{1}{T}$ and $b = 1$.

mance (before adding noise) is given by $p(\tilde{r}|\mathbf{z}, b) = \mathbb{I}(\tilde{r} = \sum_k z_k + b)$. We have that $p(s_k|\mathbf{U}, \mathbf{x}) = \mathbb{I}(s_k = \sum_i u_{ki}x_i)$ and $p(t_k|\mathbf{V}, \mathbf{y}) = \mathbb{I}(t_k = \sum_j v_{kj}y_j)$ and $p(b|\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{v}) = \mathbb{I}(b = \mathbf{x}^\top \mathbf{u} + \mathbf{y}^\top \mathbf{v})$. Therefore the joint distribution of all the variables factorizes as $p(\mathbf{s}, \mathbf{t}, \mathbf{U}, \mathbf{V}, \mathbf{u}, \mathbf{v}, \mathbf{z}, \tilde{r}, r|\mathbf{x}, \mathbf{y})$

$$= p(r|\tilde{r})p(\tilde{r}|\mathbf{z}, b)p(b|\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{v})p(\mathbf{U})p(\mathbf{V})p(\mathbf{u})p(\mathbf{v}) \cdot \prod_{k=1}^{K} p(z_k|s_k, t_k)p(s_k|\mathbf{U}, \mathbf{x})p(t_k|\mathbf{V}, \mathbf{y}).$$

The posterior distribution over the $\mathbf{U}$, $\mathbf{V}$, $\mathbf{u}$ and $\mathbf{v}$ variables given an observation, $(\mathbf{x}, \mathbf{y}, r)$, is given by summing out the latent variables: $p(\mathbf{U}, \mathbf{V}, \mathbf{u}, \mathbf{v}|r, \mathbf{x}, \mathbf{y}) \propto$

$$\int_{\mathbf{s}}\int_{\mathbf{t}}\int_{\mathbf{z}}\int_{\tilde{r}} p(\mathbf{s}, \mathbf{t}, \mathbf{U}, \mathbf{V}, \mathbf{u}, \mathbf{v}, \mathbf{z}, \tilde{r}, r|\mathbf{x}, \mathbf{y})\, d\mathbf{s}\, d\mathbf{t}\, d\mathbf{z}\, d\tilde{r}.$$

This inference is performed by approximate message passing on the equivalent factor graph [Kschischang, Frey, and Loeliger, 2001]. We assume a full factorization of the joint distribution and minimize an α-divergence (a generalization of the Kullback-Leibler divergence) between the true and approximate marginals [Minka, 2005]. We approximate all factors by Gaussian distributions. The factor graph for this model and details of all the message calculations and update schedule are given in Stern, Herbrich, and Graepel [2009].

Once the posterior marginals for the variables $\mathbf{U}$, $\mathbf{V}$, $\mathbf{u}$ and $\mathbf{v}$ have been calculated, we can discard all of the messages and continue, using this posterior as the prior for the variables for the next rating. In this way we obtain a one-pass, on-line algorithm with low memory overhead.

Application and Evaluation

A Time-Based Feedback Model In order to evaluate Matchbox for expert portfolio selection we consider two domains in which the experts are algorithms and the performance feedback is run-time. For each each task, there is a *time-out*, T, after which the algorithm is terminated and the problem is not solved. Typically, the distribution over the time it takes for algorithms to solve hard combinatorial problems has a long tail [Gomes et al., 2000]. With a modest time-out this means many problems are solved quite quickly but also many reach time-out with the remaining solution times being roughly evenly distributed between the time-out and the peak of fast solutions. To handle this, we model run time as a uniform distribution given the latent *rank, l*, of the performance of the algorithm on this task leading to a piecewise constant predictive distribution for time:

$$\begin{aligned} p(t|l=1) &= \mathbb{I}(t=T) && \text{(time out)}, \\ p(t|l=2) &= \begin{cases} \frac{1}{T-t_c} & \text{if } t_c < t < T \\ 0 & \text{otherwise} \end{cases} && \text{(slow)}, \\ p(t|l=3) &= \begin{cases} \frac{1}{t_c} & \text{if } t < t_c \\ 0 & \text{otherwise} \end{cases} && \text{(fast)}, \end{aligned} \qquad (1)$$

where t_c is the (fixed) cut-off time between a 'fast' solution and a 'slow' solution, typically $t_c \ll T$. If $l = 1$ the algorithm reached time out, if $l = 2$ the algorithm solved the problem slowly and if $l = 3$ the algorithm solved the problem quickly.

The mapping from rank to latent performance may not be linear and this mapping can change from solver to solver. We relate the latent performance r to ranks l via a cumulative threshold model [Chu and Ghahramani, 2005; Stern, Herbrich, and Graepel, 2009]. For each solver, v, we maintain solver-specific thresholds $\mathbf{h}_v \in \mathbb{R}^2$ which divide the latent rating axis into three consecutive intervals $(h_{v(i-1)}, h_{v(i)})$ each of which representing the region in which this solver attains a performance in the same rank. Formally, we define a generative model of a ranking as

$$p(l=a|\mathbf{h}_v, r) = \begin{cases} \mathbb{I}(r < \tilde{h}_{v0})\mathbb{I}(r < \tilde{h}_{v1}) & \text{if } a = 1 \\ \mathbb{I}(r > \tilde{h}_{v0})\mathbb{I}(r < \tilde{h}_{v1}) & \text{if } a = 2 \\ \mathbb{I}(r > \tilde{h}_{v0})\mathbb{I}(r > \tilde{h}_{v1}) & \text{if } a = 3 \end{cases},$$

where $p(\tilde{h}_{vi}|h_{vi}, \tau) = \mathcal{N}(\tilde{h}_{vi}; h_{vi}, \tau^2)$, and we place an independent Gaussian prior on the thresholds so $p(h_{vi}) = \mathcal{N}(h_{vi}; \mu_i, \sigma_i^2)$. The indicator function $\mathbb{I}(\cdot)$ is equal to 1 if the proposition in the argument is true and 0 otherwise. Inference for the ordinal regression observation model is performed by approximate Gaussian Expectation Propagation (EP) message passing on the factor graph as described in detail for this model in [Stern, Herbrich, and Graepel, 2009]. Note that the marginal distribution for each solver threshold must be stored.

Utility Function Faced with a fresh task, we must decide which algorithm to apply. We assume that there is some benefit for the user in solving the task but also a cost for each additional unit of time spent. Also, we realise that the balance between these objectives may vary from user to user so we use a parametrized utility function with two parameters: b, the benefit of solving a problem and c, the cost per unit time spent. The utility function is defined as

$$u(t) = \begin{cases} b - ct & \text{if } t < T \\ -cT & \text{otherwise} \end{cases}. \qquad (2)$$

For the applications here, we represent each solver by only its identity, y, so $\mathbf{y}$ is a unit vector with a one in position y. We choose a solver to apply to a task by $\hat{y} = \text{argmax}_y\, E\left[u(t)\right]_{p(t|\mathbf{x},y)}$.

Solver	K	Features	#Solved	Time	Utility
Matchbox	1	Basic	2060	15.6	1784
Matchbox	2	Basic	2064	14.1	1797
Matchbox	3	Basic	2052	13.1	1777
Matchbox	1	Combo	1978	13.5	1629
Matchbox	2	Combo	1890	17.8	1442
Matchbox	3	Combo	1818	19.8	1294
Matchbox	1	All	2052	13.0	1777
Matchbox	**2**	**All**	**2102**	**13.7**	**1873**
Matchbox	3	All	2032	15.9	1728
QuBE6.1	-	-	1966	14.3	1603
AQME	-	All	1818	11.1	1320
Oracle	-	-	2240	12.8	2150

Table 2: Batch training QBF solver portfolio performance - number solved, time per task and total utility for $c = \frac{1}{T}$ and $b = 1$. The model is first trained on the training data and then used to select which solver to use for each problem in the test problems. One solver is tried for each test problem. The performances of AQME, the best individual solver (QuBE 6.1), and the Oracle portfolio solver are included.

Solver	E	R	Se	Sk
2clsQ			$\surd$	
Nenofex	$\surd$			
QMRES		$\surd$		
Quantor	$\surd$	$\surd$		
QuBE 3.0			$\surd$	
QuBE 6.1		$\surd$	$\surd$	
sKIZZO				$\surd$
ssolve			$\surd$	
yQuaffle			$\surd$	

Table 3: Main techniques used by QBF solvers in the data-set (E=Expansion, R=Resolution, Se=Search, Sk=Skolemization)(see e.g.[Pulina and Tacchella, 2009]).

Application 1: Constraint Solving Quantified Boolean Formulas (QBF) are a powerful logical formalism to represent real-world problems, and a range of QBF solvers exist to tackle problems represented in this language [Samulowitz, 2008]. Due to the vast computational complexity of this problem, solvers display a large discrepancy in performance which is mainly due to different heuristic choices or employed solving techniques. Pulina and Tacchella [2009] introduced a self-adaptive version of an algorithm portfolio (AQME) to increase the robustness of QBF solving. AQME is self-adaptive in the sense that a classifier is retrained offline based on the information collected during execution. After finishing a run this new data point is added to the initial training set and the policy of the portfolio is updated accordingly to adapt to non-stationary data and increase the size of the training set.

We consider the following 11 QBF solvers: 2clsQ, Nenofex, QMRES, Quantor, QuBE (3.0 & 6.1), sKIZZO, ssolve (A, B, & C), and yQuaffle [Pulina and Tacchella, 2009]. Table 3 broadly characterizes each solver by the main techniques employed in QBF solving. The time-out

Solver	Pre-train	t_l	#Solved	Time	Utility
Matchbox	Yes	0	2100	14.0	1869
Matchbox	**Yes**	**10**	**2169**	**16.6**	**1996**
Matchbox	Yes	20	2172	18.8	1993
Matchbox	Yes	30	2176	21.4	1992
Matchbox	Yes	40	2174	23.2	1981
Matchbox	No	0	1960	14.3	1591
Matchbox	No	10	2139	14.9	1942
Matchbox	No	20	2156	18.9	1962
Matchbox	No	30	2163	22.0	1964
Matchbox	No	40	2161	23.9	1954
QuBE6.1	-	-	1966	14.3	1603
AQME	Yes	10	2155	18.0	1963
Oracle	-	-	2240	12.8	2150

Table 4: On-line training, 'Trust the Predicted Solver', time-out 600s, $c = \frac{1}{T}$ and $b = 1$. For this experiment the model was trained incrementally after each formula was attempted. For each problem we first allocate t_l seconds to each solver to attempt the problem, ordering the solvers by the expected utility. If none of the solvers can solve the problem in t_l seconds then we give the rest of the time before timeout to the best solver predicted by the model.

is $T = 600$ seconds and the value for t_c was set to 50s (manually tuned). We performed experiments using 2282 test problems and 288 training instances[2].

First, we show in Figure 1 a learned embedding of the solvers and problems into a 2-dimensional trait space. Matchbox was trained on the full data set with 20 iterations of message passing. The mean of the first solver trait $v_{(0)i}$ is plotted against the mean of the second solver trait $v_{(1)j}$ to produce the blue dots. The mean of the first problem trait $u_{(0)i}$ is plotted against the mean of the second problem trait $u_{(1)j}$ to produce the red crosses. Similarity in this space depends on the angle between trait vectors so similar solvers should be close in this space (in terms of angle) and this is indeed the case (except for ssolve), as shown in Figure 1.

Now we evaluate the performance of a portfolio of solvers using Matchbox and the utility function (2) for $b = 1$ and $c = \frac{1}{T}$. AQME[3] only uses a subset of the solvers, leaving out Nenofex, QMRES, QuBE 3.0 and ssolve (A,B) so we also leave out these solvers from our test set. Table 2 shows the performance of Matchbox when learning on 288 training instances and predicting a solver on the test instances without on-line adaptation. We show results for different numbers of latent dimensions K and different subsets of features (defined by Pulina and Tacchella [2009]) and the performance of AQME and the ideal portfolio manager (the oracle which always selects the fastest algorithm for each task). In terms of pure runtime we also outperform AQME (Matchbox total run-time is 111,597s versus 273,397s for AQME). We also compare the performance of the individual solvers on the

[2]The features and run-time information for each solver as well as the split in training and test data are available at `www.qbfeval.org/2008`

[3]We used AQME that employs QuBE 6.0 (not QuBE 3/5), and to our knowledge its performance is identical to QuBE 6.1.

Algorithm	K	#Solved	Time	Utility
GL	-	2531	1025	1249
CASS	-	861	207	-1768
CPLEX	-	2648	445	1672
Matchbox	1	2648	445	1672
Matchbox	2	2677	232	1804
Matchbox	3	2680	240	1807
Oracle	-	2704	248	1851

b	c	Number Solved	Mean Time
1	$\frac{1}{T}$	2677	232
1	0	2686	266
0	$\frac{1}{T}$	2667	223

Table 5: Combinatorial Auction Winner Determination. **Top:** performance of Matchbox portfolio compared with different individual solvers and the Oracle, $c = \frac{1}{T}$ and $b = 1$. **Bottom:** Comparing the effects of different utility function parameter values.

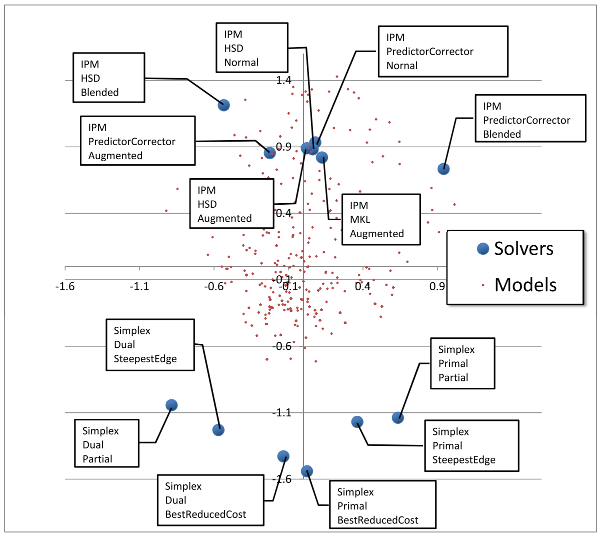

Figure 2: Learned embedding of LP solvers (compare to Figure 1). Note the separation of simplex and IPM solvers and of primal and dual approaches.

same data in Table 1.

In Table 4 we show the performance of Matchbox in the on-line setting. Here we use $T = 600s$, $c = \frac{1}{T}$ and $b = 1$, $K = 2$, and all features. We attempt each problem in two stages according to the 'Trust the Predicted Solver' (TTPS) technique [Pulina and Tacchella, 2009]: Firstly, we allocate t_l seconds to each solver to attempt the problem, ordering the solvers by the expected utility. In this round AQME sorts the solvers according to the ranking determined in the QBF competition 2006. If none of the solvers can solve the problem in t_l seconds then we enter the second round in which we give the remaining time to the best solver predicted by the model. In this way the setting of t_l allows for a trade-off between speed and robustness. The motivation for this approach is due to the long-tailed distribution of run-times: an algorithm is most likely to either solve a problem quickly or time out [Horvitz et al., 2001]. After all attempts to solve this problem we train Matchbox for the solver that solved it. In addition, if the best predicted solver times out in the second round then we update Matchbox based on this. We show results for Matchbox with and without pre-training. Pre-training means that the model was trained on the training data before running through the test data. No pre-training means the training data was not used so at the start of the run Matchbox predicts a uniform distribution over solvers. The results show that on-line adaption is beneficial and Matchbox is able to solve more instances than in the batch setting. In terms of solved instances and average run-time per solved instance Matchbox is able to outperform AQME and QuBE 6.1. The best results for Matchbox with pre-training (utility 1996) and without pre-training (utility 1964) show that it outperforms AQME (utility 1963). The time cost of training Matchbox is less than one millisecond for each problem and learning is incremental so cost does not increase with the number of problems previously seen, unlike previous work [Pulina and Tacchella, 2008].

Up to this point we have assumed that a solver must always be run from scratch each time and that solver state could not be stored in memory. Our final result for the QBF application is for the scenario where we allow dynamic switching between solvers ('suspend and resume' [Streeter and Smith, 2008]). In this case, once per second we re-calculate the expected utility for each solver, and switch to the best predicted solver at that point. In practice this yields an algorithm which behaves similarly to TTPS, because of the form of the predictive distribution of run time: after a solver is run for a time approaching t_c the expected utility for this solver drops off as a fast solution using this solver looks unlikely, leading to other solvers to be tried instead. In this way we obtain a principled approach to the scheduling problem [Streeter and Smith, 2008]. Using this method we can solve 2195 problems in the test set.

Finally, we performed a similar analysis for Linear Program (LP) solvers (see e.g. [Hooker, 2006]). The portfolio consisted of 6 variants of a simplex solver and 7 variants of an interior point method (IPM) solver from Microsoft Solver Foundation[4]. We trained Matchbox to predict run-times on a set of 4000 problems, including the Netlib LP problem set[5], among others. Figure 2 shows the latent space embedding that Matchbox learns for these solvers. Note how Matchbox separates the simplex solvers from the IPM solvers, suggesting these two techniques are suited to different types of problems. Note also that the embedding separates the primal and dual approaches along an orthogonal axis.

Application 2: Combinatorial Auction Winner Determination Next we focus on the task of determining the winner of a combinatorial auction [Leyton-Brown, Nudelman, and Shoham, 2009]. Combinatorial auctions involve self-

[4]http://www.solverfoundation.com

[5]http://www.netlib.org/lp/data/

interested agents bidding for bundles of goods while being guaranteed that the allocation of a bundle is all or nothing. The winner determination problem aims at answering the question of which bids should be accepted or not in an optimal fashion. This is a derivative of the weighted set packing problem and is therefore in general a hard computational problem (NP-hard). Leyton-Brown, Nudelman, and Shoham [2009] develop a model for determining winners in combinatorial auctions and in order to do so appropriate instances were created and a set of solvers was utilized to solve them. In addition, they define features that characterize instances and use several machine learning techniques to manage a portfolio. We used the same data[6] (run times and features) to analyze the performance of Matchbox in this domain. Table 5 (top) shows the performance of the individual solvers GL, CASS, CPLEX, the Matchbox portfolio with different numbers of latent space dimensions K, and the Oracle. The time-out T was set to 7500 seconds. The performance is shown in terms of solved instances, the mean time required per solved instance per solver, and the utility ($c = \frac{1}{T}$ and $b = 1$). When using low dimensional latent spaces, Matchbox sticks to the best single performing solver CPLEX. However, as K increases, the number of solved instances increases slightly and the average run-time is halved. Consequently, the utility of Matchbox in the best setting ($K = 3$) is better than the one achieved by any single solver (in terms of runtime and number solved) and is comparable to the one achieved by Leyton-Brown, Nudelman, and Shoham while having the advantage of incremental training. Finally, Table 5 (bottom) shows how different choices for the utility function affect the number of solved problems and the time per problem.

Conclusions

Matchbox provides a practical and scalable approach to managing a portfolio of experts. The model can learn online and dynamically adapt to non-stationary data. We allow free choice of the feedback model and utility function. The system was designed to be of general use and we see the examples here as the tip of an iceberg of applications.

Acknowledgements

We thank John Oberon and David Lao for the LP run-times.

References

Breese, J. S.; Heckerman, D.; and Kadie, C. 1998. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the 14th ACM Conference on Uncertainty in Artificial Intelligence*, 34–52.

Chu, W., and Ghahramani, Z. 2005. Gaussian processes for ordinal regression. *Journal of Machine Learning Research* 1019–1041.

Gomes, C. P., and Selman, B. 1997. Algorithm portfolio design: Theory vs. practice. In *Proceedings of the 13th ACM Conference on Uncertainty in Artificial Intelligence*, 190–197.

Gomes, C. P.; Selman, B.; Crato, N.; and Kautz, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* 24(1-2):67–100.

Hooker, J. N. 2006. *Integrated Methods for Optimization*. Springer-Verlag New York.

Horvitz, E.; Ruan, Y.; Gomes, C.; Kautz, H.; Selman, B.; and Chickering, M. 2001. A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th ACM Conference on Uncertainty in Artificial Intelligence*, 235–244.

Kschischang, F. R.; Frey, B.; and Loeliger, H.-A. 2001. Factor graphs and the sum-product algorithm. *IEEE Trans. Inform. Theory* 47(2):498–519.

Leyton-Brown, K.; Nudelman, E.; and Shoham, Y. 2009. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*.

Minka, T. 2005. Divergence measures and message passing. Technical Report MSR-TR-2007-173, Microsoft Research Ltd.

Pulina, L., and Tacchella, A. 2008. Time to learn or time to forget? strengths and weaknesses of a self-adaptive approach to reasoning in quantified Boolean formulas. In *International Conference on Principles and Practice of Constraint Programming (Doctoral Programme)*.

Pulina, L., and Tacchella, A. 2009. A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints Journal* 80–116.

Rice, J. R. 1976. The algorithm selection problem. In *Advances in computers*, volume 15. New York: Academic Press. 65–118.

Samulowitz, H. 2008. *Solving Quantified Boolean Formulas*. Ph.D. Dissertation, University of Toronto.

Smith-Miles, K. A. 2008. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.* 41(1):1–25.

Stern, D.; Herbrich, R.; and Graepel, T. 2009. Matchbox: Large scale online Bayesian recommendations. In *WWW '09: Proceedings of the 18th International Conference on World Wide Web*, 111–120. New York, NY, USA: ACM Press.

Streeter, M. J., and Smith, S. F. 2008. New techniques for algorithm portfolio design. In *Proceedings of the 24th ACM Conference on Uncertainty in Artificial Intelligence*, 519–527.

Varian, H. R., and Resnick, P. 1997. Recommender systems. *Communications of the ACM* 40:56–58.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2008. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32:565–606.

[6]http://www.cs.ubc.ca/~kevinlb/downloads.html