Minimal Model Complexity Search

Chris McConnell, ccm@cs.cmu.edu
CMU School of Computer Science
Pittsburgh, PA 15213-3891

Abstract: SURFER is an empirical discovery system that given a set of input data and a modelling vocabulary returns the model that best describes that data. The best model is considered to be the one that minimizes the description length of that model plus the data encoded using that model. The search for models is controlled by the *a priori* estimate of model likelihoods as encoded in the modelling vocabulary. SURFER includes domain independent mechanisms for identifying redundant models and for finding free parameters. The system is described together with the results of running the system on several different types of problems.

Keywords: Bayesian Learning, discovery, explanation-based learning, minimum description length.

1 Introduction

This paper describes the empirical discovery system SURFER (SURface FitER). In empirical discovery, the goal is to find a model to describe a set of observations. AI approaches to this problem are typically data-driven and use the recursive application of heuristics to generate possible models.[3] Another AI approach that has been used with some success generates models using genetic algorithms.[2] Statistical or neural network approaches use algorithmic procedures to find the parameters for a particular class of models. SURFER's design is independent of any particular modelling vocabulary and has model generation heuristics that can plausibly be learned. It is a model-driven approach based on the ideas of Minimum Description Length (MDL).[6]

What is it that makes one model better than another? One answer is a model that fits the data exactly. Unfortunately for a limited set of data there are an infinite number of models with that property. Furthermore, real world data often includes noise. The principle of *Occam's Razor* indicates that the simplest model should be preferred, but model complexity also needs to be traded off against accuracy. A model that fits noisy data exactly or that is simple but doesn't match the data very well will be unlikely to be predictive.

MDL formalizes the tradeoff between model complexity and accuracy.[6] MDL views model selection as a coding problem. Given a coding system for describing models, the best model is the one where the string describing the model plus the string describing the data in terms of the model has the shortest length. This formulation is equivalent to a Bayesian one since the coding cost of a string in bits C can be interpreted as the probability 2^{-C} of that string.[4] Under this interpretation, the complexity of a model is equivalent to its prior probability and the length of a model plus the data is the posterior probability that the model fits the data.

A model coding system for a particular problem can be considered a *domain theory* for that problem. The coding system defines a partial order based on complexity for all possible models. Furthermore, MDL rules out some models as a possiblility since the complexity of the data places a limit on the most complex model that should be considered. SURFER exploits this partial order to generate models and then evaluates them by the MDL criterion.

There have been other model-driven approaches to empirical discovery such as COPER[1] and the system built by Solomonoff.[9] COPER's domain theory made use of knowledge about units to identify likely theoretical terms and then fit a parameterized model to those terms. Solomonoff's approach is much closer to that of SURFER. Its search process is based on the ideas of Kolmogorov complexity.[8] A model is a program for generating candidate solution strings. The probability P_i of model i being the solution is defined as the product of the probabilities of the concepts making it up. All possible models are generated at once with no attempt at identifying equivalent models. The search procedure selects a time T and then spends time TP_i executing model i. If no model has found a solution, T is doubled and the search continues. Solomonoff's search procedure is theoretically elegant but presents numerous practical difficulties.

No results on this system have been published.

The SURFER system described in this paper successfully implements an empirical discovery system that generates models based on a probabilistic domain theory. The models are evaluated in terms of the MDL criterion. The system incorporates domain independent mechanisms for identifying equivalent models and for optimizing free parameters. Despite being a version of generate and test, the system is able to solve moderately difficult problems.

2 SURFER

Given a dataset and a modelling vocabulary, SURFER returns the model that minimizes the description length of the model and the data encoded using that model. Models are generated in order of increasing complexity up to the complexity justified by the dataset.

A *dataset* consists of a dataset signature and k n-tuples of data. The *dataset signature* describes the names, types, ranges and directions associated with each element in an n-tuple. Types can be *integer, boolean* or *real.* Direction is *in* for elements to be used as model arguments and *out* for elements to be predicted by models. Each *n-tuple* has a set of n values with characteristics as described by the signature.

A vocabulary for SURFER consists of a set of terms. Each term can either be a terminal or an operator for combining other terms. Each term has a name, type, relative cost and class. The *type* of a term indicates the type of value the term represents in an expression: boolean, integer or real. *Relative cost* reflects the relative cost of coding a term in a model. The *class* of a term is either constant, parameter, variable or operator.

Constant, parameter and variable terms are all terminals. A *constant term* represents a constant value such as one, π or zero. A *parameter term* represents a free parameter in the model that can be optimized to fit a model to a particular dataset. A *variable term* represents one of the *in* elements in a dataset n-tuple.

Operator terms combine one or more component terms of specified types. If the order of component terms does not matter, an operator is marked as commutative. Operator terms require a fixed number of component terms, but arbitrary arity operators can be simulated by marking an operator as associative. Recursive operator terms are also marked so that the operator can refer to the entire model. Operator terms also include their identity constant and inverse operator if applicable.

Throughout this paper, models will be described in prefix notation. Constants will be represented by `c`, parameters by `p`, variables by `x`, and operators by their name. For example, the addition of a constant and a variable would be shown as: `(+ c x)`.

2.1 Models

Models are built out of the terms found in the vocabulary. The *model complexity* of a model is the number of bits required to encode that model. It is the sum of

Name	*Args*	*Identity*	*Inverse*	*Commut.*	R_t	P_t	C_t
`X`	Variable				1.0	0.22	2.2
`Float`	Parameter				1.0	0.22	2.2
`*`	Float X Float	1.0	`/`	Y	1.0	0.22	2.2
`/`	Float X Float	1.0	`*`	N	1.0	0.22	2.2
`LOG`	Float		`EXP`	Y	2.0	0.05	4.3
`EXP`	Float		`LOG`	Y	2.0	0.05	4.3

Fig. 1. Depth Complexity Terms

the costs of the terms making up that model. Once a vocabulary V is selected, the cost C_t and probability P_t of each term t can be calculated from its relative cost R_t. To do this, the base coding cost b is calculated as the solution of: $\sum_{t \in V} 2^{-R_t b} = 1.0$. The base coding cost can then be used to calculate $C_t = R_t b$ and $P_t = 2^{-C_t}$. Figure 1 shows a typical vocabulary of terms with the relative cost of unary operators doubled. With these values, models will be generated in terms of increasing model depth. The model `(* X (LOG X))` will have the same model length as the model `(* (LOG X) (EXP X))` even though the second model has one more term than the first.

The *description length* of a model is the model complexity plus the number of bits required to encode the dataset using that model. Given an n-tuple as input, a model predicts a value m. The n-tuple is encoded using a Gaussian probability distribution $G(m, fm)$ with a standard deviation of $\pm fm$ where f is the desired fractional tolerance. Given $G(m, fm)$ for an n-tuple, the number of bits required to send that n-tuple with the model is $-\log_2 G(m, fm)$.

A model can include free parameters that need to be optimized to the data. To assign free parameters, SURFER uses numerical optimization procedures to minimize the model description length. Functions with one free parameter are optimized using Brent's method. Functions with multiple parameters are optimized using Powell's method with line minimization done by Brent's method. [5] As in most numerical optimization procedures, these are prone to false minima, but they work well for the class of models currently tested with SURFER.

Model vocabularies are usually highly redundant. There are syntactic rearrangements, and parameterized equivalences. For example, Figure 2 shows some of the equivalences present in a vocabulary with some typical mathematical functions. There are several different techniques used in SURFER to handle these equivalences. Pattern 1 is handled by generating only argument combinations for commutative functions. If a term is not commutative then all argument permutations are generated. Term inverse information is used to reject inverse patterns like pattern 2. Since SURFER allows parameterized functions, patterns like 3 and 4 can arise where the parameters are redundant. These are identified by propagating parameters upward through terms that are the same as the top level term or its inverse. If there end up being more than one constant at the top level, the model is rejected. Models with more than doubly recursive functions are also rejected to keep the evaluation of signatures within reasonable bounds.

SURFER uses the domain independent, probablitistic technique of signatures to identify model equivalences. A *model signature* is the evaluation of a model on a random set of probes. Each probe binds the input variables and free parameters to random values. Functions with singularities like division, generate special values at the singularities that contaminate all other operations so that the signature value for the probe causing the singularity is marked as undefined. If the top level term is recursive, no signature is generated since the component's value will depend on the model it is embedded in. If all of the elements in a signature are constant as in pattern 5, the model is rejected. Otherwise it is compared to previously generated models. If there is a match, then the model with the greatest complexity is rejected as in patterns 6 and 7.

Once a model's parameters have been fixed, each model component with a free parameter is checked to see if its parameter is close to the identity constant for that component. If that is the case, the model is kept because the identity constant would cause that model component to have no effect whereas a different parameter value would have an effect. Pattern 8 shows an example with an identity constant. If no components have identity constants, the same random probes used for signatures are used to generate an *instance signature*. The instance signature is then used to find other models with free parameters that converged to the same function. Pattern 9 shows an equivalence caught by instance signatures.

Signatures for integer and boolean valued models must match exactly and are checked using hash tables. Floating point models require special care due to the imprecise nature of floating point calculations. Two signatures are considered the same if each element of the signature is within a fractional tolerance of the value or within an absolute value of zero. *KD-trees* [7] are used to implement this matching efficiently.

The probability of a false match by a signature with b bits per signature and k probes in the signature is 2^{-bk}. In SURFER, models with float outputs usually have $b = 22$ and $k = 5$, so the likelihood of a false match is approximately 1 in 10^{34}. Models with integer outputs have $b = 8$ and $k = 10$ so they have an error of 1 in 10^{24}. Models with boolean outputs have $b = 1$ and $k = 20$, so they have an error of 1 in 10^{20}. The error rate is actually higher than these values would indicate since most functions are relatively smooth. With smooth functions, being close at one point increases the likelihood that the functions will be close at nearby points. To help prevent this effect the random probes are constructed to insure the presence of very small and large positive and negative values for each parameter and variable.

The matching procedure used for floating signatures and instance signatures can have both false positives and false negatives. False positives are bad because they permanently remove a unique model. False negatives are not as bad, but they do cause more models than are necessary to be generated. The incidence of false positives is decreased by verifying matches with an additional probe. If the models do not match on the additional probe, then the match is rejected. In practice, false positive signature matches happen very rarely. False positives

happen much more frequently for instance signatures because of the inherent numerical problems involved in optimization. Tightening the tolerances used for matching instance signatures decreases the false positive rate, but increases the false negative rate.

Despite these problems, the signature matching works well in practice. The only problem that has false positives is the *Newton* problem as described in Figure 3. In the 8,361 models generated there were no false positives for signatures, but there were 23 for instance signatures, all of them with more complicated models and frequently involving two free parameters.

```
1. (+ X1 X2) == (+ X2 X1)
2. (LOG (EXP X)) == X
3. (+ P1 (+ X P2)) == (+ X P3)
4. (+ P1 (- X P2)) == (+ x p3)
5. (/ X X) == 1
6. (+ X (- X X)) == X
7. (* X (+ C1 X)) == (+ (* C1 X) (* X X))
8. (+ (* X 1.0) X) == (+ X X), (+ (* X 1.5) X) != (+ X X)
9. (* X P1) == (/ X P2)
```

Fig. 2. Redundant Patterns

2.2 Generation

SURFER generates models in order of increasing complexity. At any given time there are usually many possible models with the same complexity. All of these models are generated before any more complex models are considered. A *model complexity level* consists of all the models with the same complexity that are kept after being generated. All higher model complexity levels are constructed out of components from lower levels. At a given complexity level, the arguments for unary terms all come from the same lower level. For higher arity functions the picture is more complicated. A binary term might have the same complexity models with a level zero component and a level three component as with two level two components.

Each time the system completes a model complexity level, all of the possible model complexity level combinations for each possible term arity are generated. These combinations are then inserted into a list for each arity ordered by complexity total and when the complexity totals are equal, ordered by the smallest maximum complexity level involved. Once a term has generated all possible combinations (or permutations for non-commutative functions) of the models in a given combination of model complexity levels, it moves onto the next combination in the list for its arity.

The first model generated is always a terminal (constant, variable or parameter) since any non-terminal model must include one or more terminals. On each

cycle, a term has associated with it pointers to the previously kept models it would like to use as components, the complexity of the candidate model and an accumulated probability. At each stage only terms of the problem output type and with minimal complexity are considered. Within those terms the first term with an accumulated probability greater than one is generated and has its probability decremented by one. If there are no terms with an accumulated probability greater than one, every vocabulary term is incremented by its associated probability P_t. This procedure has the effect of insuring that models are generated in order of increasing complexity while causing terms to appear as often as the top level operation in a model as their probability P_t.

Operators can overlap. For example, both `(* x x)` and `(* x1 x2)` can coexist as terms in the same vocabulary even though the first is just a special case of the second. Presumably `(* x x)` is used enough that using some of the coding space for the special case results in shorter model solutions. Since models are searched in order of complexity, such terms change the model generation order.

Figure 4 shows the model generation process for a simple problem. The problem is defined by the *Example* entry in Figure 3. `*` has a relative cost R_t twice that of the other terms, so the complexity C_t of `*` is 3.44 and C_t of the other terms is 1.72. *(Cy)cle* is incremented each time a model is generated. *(K)ept* is incremented each time a model is kept. *(R)ejects* keeps track of the number of models that were rejected during model generation as were models 4, 7 and 11. *(S)ignatures* is the number of models rejected because their signature matched a previously generated model. Model 10 was rejected because its signature was the same as model 9. *(I)signatures* is the number of models that were rejected after their free parameters were optimized to the data. Model 13 was rejected because its instance signature was the same as that for model 6. *(Ev)aluations* is the number of evaluations of the model the optimization procedures took to optimize the free parameters in the model. *(Co)mplexity* is the model complexity in bits. As can be seen in the figure five complexity levels were generated in order of increasing model complexity. *(Len)gth* is the total description length of the model. The end of each line shows the model just generated. If the model is the best so far it is marked with an @ sign. If the model is rejected, an = is printed and its equivalent model is shown. In this example, the original model is found on model 14 of 20 possible models.

3 Results

Figure 3 shows some of the problems that SURFER has solved. *Boolean* looks for a simple boolean function of four variables. The problems *Regression* and *Fibonacci* come from work on genetic programming.[2] The *Regression* problem requires finding both the symbolic form of a model and the parameters required to make the model fit the data. The *Fibonacci* problem requires finding a recursive program to generate element `J` of the Fibonacci sequence. The `SELF` function returns one for any value less than two and zero if the function would recurse infinitely. *Newton* is just a more complex instance of the regression problem

Name	*Inputs*	*Output*	*Const.*	*Unary*	*Binary*
	Expression				
Example	X	Float			+ *
	(+ 3.0 (* X X))				
Boolean	A B C D	Boolean		NOT	AND OR
	(AND (OR A B) (OR C D))				
Regression	X	Float		- COS SIN EXP LOG	+ * /
	(+ (* 2.718 (* X X)) (* 3.1416 X))				
Fibonacci	J	Integer	0 1 2	SELF	+ -
	(+ (SELF (- J 2)) (SELF (- J 1)))				
Newton	M1 M2 R	Float			+ * /
	(/ (* 6.67 M1 M2) (* R R))				

Fig. 3. Problem Definitions

```
Cy   K R S I Ev Co Len Model
00:  0 0 0 0  0  2  98 @X
01:  1 0 0 0 28  2  52 @FLOAT1
02:  2 0 0 0  0  5  98  (+ X X)
03:  3 0 0 0 12  5  54  (+ X FLOAT1)
04:  3 1 0 0  0  5      (+ FLOAT1 FLOAT1) =
05:  4 1 0 0  0  7  54  (* X X)
06:  5 1 0 0 13  7  92  (* X FLOAT1)
07:  5 2 0 0  0  9      (* FLOAT1 FLOAT1) =
08:  6 2 0 0  0  9  97  (+ X (+ X X))
09:  7 2 0 0 12  9  64  (+ X (+ X FLOAT1))
10:  7 2 1 0  0  9      (+ FLOAT1 (+ X X)) = (+ X (+ X 3.00))
11:  7 3 1 0  0  9      (+ FLOAT1 (+ X FLOAT1)) =
12:  8 3 1 0  0 10  58  (+ X (* X X))
13:  8 3 1 1 12 10  95  (+ X (* X 3.00)) = (* X 4.00)
14:  9 3 1 1 11 10  10 @(+ FLOAT1 (* X X))
(+ 3.0143678 (* X X))
```

Fig. 4. Example Problem

where the goal is to discover Newton's universal law of gravitation including the constant. To prevent numerical problems, the constant used is not of the right magnitude.

Each of these problems was run with two different sets of relative term complexities except *Newton*. In the first run, all terms have the same relative complexity $R(t)$. In the second run, unary functions have a relative complexity R_t twice the complexity of the other terms. Since *Newton* has no unary terms only one run is provided because the results would be the same in both cases.

In the first run, the complexity level of a model is just the number of terms in that model minus one. In the second run, the complexity level of a model is one less than the number of terms in that model minus the number of binary terms. The model `(SIN X)` would be found at model complexity level one in both

runs. Model `(+ X (+ X C))` would be found at model complexity level four in the first run and two in the second run. Unless relative complexities are integer multiples of the lowest complexity terminal, determining a model's complexity level is usually not this simple.

Figure 5 shows the best and second best model for each problem run and their respective model complexities and description length. On each run, SURFER ran until it found a model that matched the data exactly. This was just an expedient way to get each run to terminate quickly. In the presence of noise, each run would have to continue to either a preset effort limit or until models get complex enough that they cannot improve on the best description length. As can be seen by the large difference between the best and second best model in each case, there would have to be a lot of noise or very few n-tuples to cause the system to prefer the second best model. As can be seen in the *Regression* problem solutions, the two runs do not always end up generating the same syntactic model although they do generate models that describe the same function.

Problem	*Best* *Second Best*	*Complex (Bits)*	*Len (Bits)*
Regression	`(* (+ x 1.15) (/ X 0.37))`	23.3	23.3
	`(* (+ X X) (+ X 1.21))`	23.3	44.5
Regression2	`(* (* X 2.7) (+ X 1.2))`	18.0	18.0
	`(* (+ X X) (+ x 0.8))`	18.0	37.0
Boolean	`(AND (OR D C) (OR A B))`	19.7	19.7
	`(OR D C)`	8.4	27.2
Boolean2	`(AND (OR D C) (OR A B))`	18.4	18.4
	`(OR D C)`	7.9	26.6
Fibonacci	`(+ (SELF (- J 1)) (SELF (- J 2)))`	25.3	25.3
	`(+ (- J 1) (SELF (- J 1)))`	22.5	55.8
Fibonacci2	`(+ (SELF (- J 1)) (SELF (- J 2)))`	28.9	28.9
	`(+ (- J 1) (SELF (- J 1)))`	23.6	124.0
Newton	`(* (/S 6.7 R) (/S (* M1 M2) R))`	25.3	25.3
	`(* (/S 0.09068619 R) (* M1 M2))`	19.7	35.9

Fig. 5. Problem Solutions

Figure 6 shows information on the search process for each problem. *Kept* is the number of models that were kept of all of the models *Tried*. *Max* is the maximum number of models that would have to be tried to guarantee finding the eventual solution. It is the number of models that were left in the same complexity level as the solution. *Generation* is the number of models that were rejected without being evaluated. *Signature* is the number of models that were rejected as being identical to a previously kept model through the use of signatures. *Instance* is the number of models that were rejected as being equivalent to another once their free parameters were optimized to fit the data. *Evals* is the number of model evaluations that were required including the optimization procedures.

Since *regression* and *Fibonacci* came from work on Genetic Programming, *GP Evals* has the number of evaluations Genetic Programming requires.[2]

Problem	*Models*			*Rejects*			*Evals*	*GP*
	Kept	*Tried*	*Max*	*Gen*	*Sign*	*Inst*		*Evals*
Regression	4519	6121	33224	378	854	370	36184	306,000
Regression2	141	194	1584	20	17	16	1851	306,000
Boolean	332	926	1520		594		926	
Boolean2	332	926	1520		594		926	
Fibonacci	5626	6997	21974	1070	301		6997	484,500
Fibonacci2	5169	6421	73328	316	914		6421	484,500
Newton	4095	8631	62661	323	3166	1047	133106	

Fig. 6. Problem Summary

As can be seen in Figure 6, the solution to *Regression* was found much sooner in the second run than in the first. This is because the solution only involved binary functions and the change in relative complexities caused binary functions to appear at earlier complexity levels. However even if the order of models within the same complexity level had been different, the worst case for the first run was 33,224 models while that for the second run was only 1,584.

Boolean looks like the two runs had exactly the same effect, but the path taken to the results are actually very different. The number of complexity levels explored for the first run was four and for the second was seven. The reason that the size of the final complexity levels are so similar is because of the very high redundancy found in the boolean terms.

While problem *Fibonacci* looks similar to *Regression* in that the second run finds a model sooner than the first, that is actually deceptive. The maximum number of models that would have to be generated in the first run is only 21,974 whereas in the second run it is 73,328. It was only being lucky that made it look like run two found a solution more quickly than the first run. The reason that the effect of the two runs is different from their effects in *Regression* is that the solution to *Fibonacci* involves unary functions. The second run causes the size of the model complexity levels to increase because unary functions must be generated for all of the binary functions at the previous model complexity level. Since the final solution must await the generation of all the models at the level of the unary components, the size of the final model complexity level is much larger.

While *Fibonacci* has the runs with the largest final model complexity level and the most number of model complexity levels, *Newton* is the problem that takes the longest to solve. This is because of the large size of the complexity levels and the effort spent finding optimal settings for parameterized models.

How do *Regression* and *Fibonacci* compare to Genetic Programming (GP)? As can be seen in Figure 6, GP requires substantially more evaluations on both of these problems. This is because GP works best when successful building blocks

can be found and then combined. In these examples, there are not small successful building blocks and GP does not do very well when compared to SURFER.

4 Conclusion

SURFER as it stands is a useful system for solving some empirical discovery problems. It is a framework with a single unifying idea–that of generating models in increasing complexity order. It provides domain independent mechanisms for model generation, optimization and testing. If a problem has the additive property that partial models can be identified and then combined, data-driven systems would be better. In systems without this property, but with a good *a priori* idea of likely model components, SURFER would be a good choice. The main advantages of this architecture are its complete coverage of the model space and that the domain theory is amenable to automatic generation. I plan to extend the system to automatically refine the domain theory after solving several similar problems by adjusting the vocabulary to incorporate new terms or to adjust the prior probablilities of the terms. This addition will allow the system over time to more quickly find optimal models in a particular domain.

References

1. M. M. Kokar. Determining arguments of invariant functional descriptions. *Machine Learning*, 1:403–422, 1986.
2. John R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Stanford University, 1990.
3. Pat Langley and Jan M. Zytkow. Data-driven approaches to empirical discovery. *Artificial Intelligence*, 40(1):283–312, 1989.
4. John R. Pierce. *An Introduction to Information Theory.* Dover Publications, Inc, New York, NY, 1980.
5. William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, England, 1988.
6. J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.
7. Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, New York, 1990.
8. R. J. Solomonoff. A formal theory of inductive inference. parts i and ii. *Information and Control*, 7:1–22, 224–254, 1964.
9. R. J. Solomonoff. A system for incremental learning based on algorithmic probability. In *AAAI Symposium on the Theory and Application of Minimal-Length Encoding*, pages 140–146, March 1990.

This article was processed using the LaTeX macro package with LLNCS style