Journal of Machine Learning Research ? (2010) ?-? Submitted 7/10; Published ?/10

Predicting Student Performance: A Solution for the KDD Cup 2010 Challenge

Yongming Shen, Qiong Chen, Ming Fang, Qiuyong Yang, Tingting Wu, Lixiong Zheng, Zongfa Cai
South China University of Technology SYMINGZ@FOXMAIL.COM

Editor: ?

Abstract

This paper describes our solution for the KDD Cup 2010 Challenge. The solution is based on a solution framework designed by us. And the framework is a general data-mining procedure that is able to handle asymmetric training and test sets as well as non-atomic and variable-length attributes. The paper will first present the solution framework, and then the solution that we used to generate the final submission.

1. Introduction

The KDD Cup 2010 challenge is about predicting student performance on mathematical problems from logs of student interaction with intelligent Tutoring Systems. Data sets used in the challenge come from multiple schools over multiple school years, and Tutoring Systems used include the Carnegie Learning Algebra system and the Bridge to Algebra system.

In our opinion, the prediction task is challenging in two aspects. First, training and test sets are Asymmetric. Besides the class attribute, there are other attributes that are missing in test sets. What's more, the temporal relation between a training set and its matching test set may make them exhibit different statistical properties. If this is indeed the case, it would be a problem, as many classification algorithms expect the training set and test set to be statistically similar. With such asymmetries between training and test sets, sophisticated data preprocessing is needed. Second, there are non-atomic and variable-length attributes. Both KC and Opportunity are non-atomic and variable-length attributes, this also requires special preprocessing.

The remainder of this paper is organized as follows. Section 2 presents the solution framework. Section 3 is a detailed description of the solution that we used to generate the final submission. Section 4 is the result and summary. Appendix A presents other things that we have tried.

2. Solution Framework

This section describes a solution framework designed by us. The framework was settled at the very beginning of our work. So basically, most of our time was spent trying out different applications of this framework and finding the best ones. The framework can also be looked at as the big picture of its applications.

2.1 Motivation

As pointed out in Section 1, the data sets of KDD Cup 2010 have asymmetric training and test sets as well as non-atomic and variable-length attributes. Thus the traditional data-mining procedure, which assumes symmetric training and test sets and atomic attributes, might not work well here. And so we figured we should design a new way to handle the data sets, that is, a new solution framework.

2.2 Design

The design of our framework was inspired by a simple idea: if a student performed well in the past and the step to be performed is simple, chances of the student making a correct first attempt will then be high. With this in mind, a vague prediction procedure was developed: First, build a scoring machine from the training set. The scoring machine should return a score vector when given a test record (that is, a test set step record). And scores in the returned vector should reflect Correct First Attempt relevant info like student performance, step difficulty, etc. Second, use the scoring machine to compute a score vector for every test record. Third, make predictions from the score vectors.

To make the above procedure concrete, we need to solve two sub-problems, that is, how to build a scoring machine, and how to make predictions from score vectors. Following are our solutions to these two problems.

2.2.1 Build a Scoring Machine

A simple scoring machine could be built like this: First, use the training set to build two dictionaries. The first dictionary should use Anon Student Id values as keys, and each key should be associated to its corresponding student's average Correct First Attempt. The second dictionary should use Step Id values (a record's Step Id equals its Problem Hierarchy+Problem Name+Step Name) as keys, and each key should be associated to its corresponding step's average Correct First Attempt. Second, make a scoring procedure from the two dictionaries. When given a step record, the procedure should first use that record's Anon Student Id to look up a score from the first dictionary, and then use that record's Step Id to look up another score from the second dictionary, and finally return the score vector formed by those two scores. Third, return the scoring procedure as the scoring machine.

Scoring machines built according to the above algorithm will apparently be over simplified, as many attributes available in the training set are not used at all. One obvious way to enhance this algorithm is to make it build more dictionaries. And this is exactly what we did when designing practical algorithms for our solutions. Details of some of the practical algorithms will be given in Section 3 and Appendix A.

Dictionary based scoring machines are not dictated by the framework, and thus we can also build scoring machines using some racially different methods. We will not discuss such methods in the paper, as we haven't tried any of them. In fact, as dictionaries are quite expressive, dictionary based scoring machines can possibly simulate all other kinds of scoring machines.

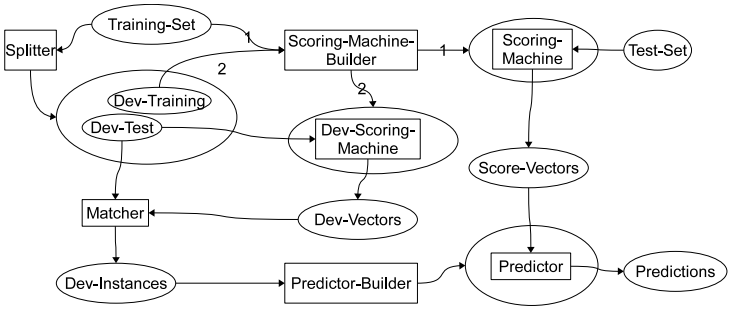

Figure 1: The data-flow diagram of our solution framework.

2.2.2 Make Predictions from Score Vectors

To make predictions from score vectors, we will need to build a predictor, that is, a function which takes a score vector as input and returns a probability prediction as output. As a score vector can have many elements, building such a function heuristically will be impractical, and thus a systematic way should be used instead. In situations like this, machine learning makes a natural solution. But to use machine learning methods, training instances are needed, which we don't have. To get training instances, we will need a data set which is similar to the one we are working on, and yet has no unknowns. The solution to this is apparent: such a data set can be obtained from the training set.

And here are the steps for making predictions from a collection of score vectors (call it Score-Vectors): First, split the training set into two parts. Let's call them Dev-Training and Dev-Test. Second, build a scoring machine from Dev-Training. The machine building algorithm should be the same as the one used to generate Score-Vectors. Third, compute a score vector for each step record in Dev-Test, and call the resulting vectors Dev-Vectors. Fourth, match every vector in Dev-Vectors with its Correct First Attempt in Dev-Test. Each matched pair will be a training instance for the predictor. Call the resulting collection of training instances Dev-Instances. Fifth, feed Dev-Instances to some machine learning algorithm to train a predictor. And finally, use the trained predictor to make predictions from Score-Vectors.

As for how to split the training set, Section 3 and Appendix A will give some examples, so we will leave it for now.

2.3 Data Flow

To offer a better view of the solution framework as a whole, the data flow of the framework in presented in this section.

Figure 1 is the data-flow diagram of the framework. In the diagram, an ellipse represents a piece of data; a rectangle represents a function; an arrow pointing from an ellipse to a rectangle represents a function application; an arrow pointing from a rectangle to an ellipse represents a function return; if a function is applied more than once, different application-return arrow pairs will be marked by different pair labels; if one entity is inside of another entity, it means the former is a component of the latter; every function is pure, that is, an output of a function will only depend on its corresponding input.

From the diagram we can see the data flow begins from Training-Set (that is, the training set) and Test-Set (that is, the test set), and ends in Predictions. Since functions are pure, middle steps can be carried out in any order, so long as data dependencies are respected.

We can also see that besides Training-Set and Test-Set, four more pieces are needed to get things going, and they are Splitter, Scoring-Machine-Builder, Matcher and Predictor-Builder. According to the framework's design, only Matcher is fixed, so we can consider the other three as parameters of the framework. And since everything else (that is, everything other than Training-Set, Test-Set and the four pieces) in the diagram is derived, these three parameters are also the only parameters. Thus to design a solution based on this framework, one will only need to provide an implementation of these three pieces. Section 3 will give a detailed example of one such implementation.

3. Final-Submission Solution

This section will describe our final-submission solution in detail. As the solution is an application of the solution framework, the presentation will naturally consist of three parts, corresponding to the three parameters of the framework. Following the presentation, we will also give some general descriptions about the solution's time and space performance, and its software implementation. In following subsections we will simply call the final-submission solution "our solution" or "the solution".

3.1 Splitter

The splitter of our solution was designed according to this principle: the relation between Dev-Training and Dev-Test should be similar to the relation between Training-Set and Test-Set. The reason for this is that the resulting training instances may thus resemble the test instances better.

And here is how the splitter works: First, divide Training-Set into blocks. If two records have the same Anon Student Id and Unit Id (Unit Id means the unit part of Problem Hierarchy), they go to the same block, and otherwise they go to different blocks. Records in a block should be in ascendant order with respect to their Row's. Second, divide each block into two parts. One part should contain records of the block's last problem, and the other part should contain the rest. Third, collect the last problem parts to form Dev-Test, and the other parts to form Dev-Training.

To summarize, with the above procedure being used, Dev-Training is simply Training-Set with all "block last problems" removed, and Dev-Test is simply the difference between Training-Set and Dev-Training. Note that the size of Dev-Test will be comparable to that of Test-Set, and so Dev-Instances, which has the same size as Dev-Test, will be rather small compared with Training-Set, and this contributes a lot to our solution's time performance.

3.2 Scoring-Machine Builder

Within our solution, two different scoring-machine builders are used, one for each challenge data set. This two machine builders are generated by a meta-builder, which takes a collection of dictionary schemes and returns a scoring-machine builder.

The following two paragraphs explain how the meta-builder works.

A dictionary scheme consists of two parts, a key scheme and a score scheme. A key scheme determines how a key is generated from a step record. A primitive key scheme is denoted by a single attribute name, and it uses the corresponding attribute values as keys. A compound key scheme is a set of primitive key schemes, and it uses the corresponding primitive key sets as keys. A score scheme determines how to compute a score from a key, a key scheme and a step-record collection. To generate a dictionary from a dictionary scheme and a step-record collection, run the key scheme through the collection to get the set of keys, and then use the score scheme to compute a score for each key in the set. To look up a score in a dictionary for a step record, use that dictionary's key scheme to find the record's key, and then find the score of that key.

Let SMB be a scoring-machine builder made by the meta-builder using a dictionary-scheme collection DSC. When given a step-record collection SRC, SMB will build a scoring machine like this: First, use DSC and SRC to generate a collection of dictionaries. Second, make the scoring procedure. When given a step record, the procedure should use that record to look up a collection of scores from the collection of dictionaries, and then return the score collection as the score vector. Third, return the scoring procedure as the scoring machine.

Now we know how the meta-builder works, and so to present the scoring-machine builders we use, only their corresponding dictionary-scheme collections need to be given.

3.2.1 Some Notation

As the size of a dictionary-scheme collection can be quite large, a special notation is introduced to facilitate subsequent presentations. Following is a listing of the constructs of this notation and their meanings.

K0+K1
A compound key scheme formed by combining key schemes K0 and K1.

$
The empty key scheme. $+K = K+$ = K.

K:S
A dictionary scheme with key scheme K and score scheme S.

{K0, K1}:*{S0, S1}
Shorthand for the collection {K0:S0, K0:S1, K1:S0, K1:S1}. In the general case, the key-scheme collection and the score-scheme collection can have arbitrary sizes, and a dictionary-scheme collection is generated by first compute their product, and then turn pair elements of the product into corresponding dictionary scheme elements.

{K0, K1}+*{K2, K3}
Shorthand for the collection K0+K2, K0+K3, K1+K2, K1+K3. Can be generalized like the ":*" construct.

And here are some examples of the notation:

{K}:*{S0, S1, S2} means {K:S0, K:S1, K:S2}

{K0}+*{K1, K2}:*{S} means {K0+K1:S, K0+K2:S}

{K0}+*{K1, K2}:*{S0, S1} means {K0+K1:S0, K0+K2:S0, K0+K1:S1, K0+K2:S1}

{K0+K1}+*{K2, K3} means {K0+K1+K2, K0+K1+K3}

{K0, $}+*{K1, K2}+*{K3} means {K0+K1+K3, K0+K2+K3, K1+K3, K2+K3}

3.2.2 Dictionary-Scheme Collections

The dictionary-scheme collection for Algebra 2008-2009 contains 103 dictionary schemes. Following is a listing of them.

Dictionary Schemes 1 to 7
{Anon Student Id, Unit Id, Section Id, Problem Id, Step Id, Block Id, Anon Student Id Section Id}:*{ Expected Correct First Attempt}

Dictionary Schemes 8 to 91
{$, Anon Student Id, Unit Id, Section Id, Problem Id, Block Id, Anon Student Id+Section Id}+*{KC(SubSkills), KC(KTracedSkills), KC(Rules)}:*{KC Expected Corrects, KC Expected Incorrects, KC Expected Hints, KC Expected Correct First Attempt}

Dictionary Schemes 92 to 103
{Block Id}+*{KC(SubSkills), KC(KTracedSkills), KC(Rules)}:*{KC Damp Corrects, KC Damp Incorrects, KC Damp Hints, KC Damp Correct First Attempt}

The dictionary-scheme collection for Bridge to Algebra 2008-2009 contains 31 dictionary schemes. Following is a listing of them.

Dictionary Schemes 1 to 7
{Anon Student Id, Unit Id, Section Id, Problem Id, Step Id, Block Id, Anon Student Id Section Id}:*{ Expected Correct First Attempt}

Dictionary Schemes 8 to 23
{$, Anon Student Id}+*{KC(SubSkills), KC(KTracedSkills)}:*{KC Expected Corrects, KC Expected Incorrects, KC Expected Hints, KC Expected Correct First Attempt}

Dictionary Schemes 24 to 31
{Block Id}+*{KC(SubSkills), KC(KTracedSkills)}:*{KC Damp Corrects, KC Damp Incorrects, KC Damp Hints, KC Damp Correct First Attempt}

And following are explanations for some of the names used in the dictionary-scheme listings.

About the key schemes, Unit Id is the unit part of Problem Hierarchy; Section Id is the Section part of Problem Hierarchy; Problem Id means Problem Hierarchy+Problem Name; Step Id means Problem Id+Step Name; Block Id means Anon Student Id+Unit Id.

About the score schemes, see the listing that follows.

Expected Correct First Attempt
This scheme computes a score as follows: First, find all key-matching records in the step-record collection, excluding those with missing Correct First Attempt. Second, compute the average Correct First Attempt of the selected records, and return it as the score.

KC Expected Correct First Attempt
This scheme only works with key schemes having the form "k+KC(t)" ("k" and "t" are variables, "k" can be any key scheme; "t" can be SubSkills, KTracedSkills or Rules). And here is the procedure of this scheme: First, split the key into two parts, with one part corresponding to "k" (call it k-Key) and the other to "KC(t)" (call it KC-Key). Note that this implies KC-Key will be a knowledge component list with "∼∼" as the element separator. Second, compute a score for every knowledge component in KC-Key. A knowledge component score is computed by first finding all records matching k-Key, excluding those with missing Correct First Attempt and those don't contain the knowledge component in question, and then calculate the average Correct First Attempt of the selected records and use it as the score. Third, find the average of all knowledge component scores and return it as the final score.

KC Damp Correct First Attempt
This scheme is nearly the same as KC Expected Correct First Attempt, except that in the second step, instead of average Correct First Attempt, $\sum_{i=1}^{|V|} V_{[i]} 0.8^{|V|-i}$ is used, where V is an array containing all the selected Correct First Attempt values (values are ordered according to their Row, a smaller Row means a smaller index), and $|V|$ is the size of V.

Others
Meanings of other score schemes can be obtained by generalizing the above three explanations appropriately.

3.2.3 Some discussion

From the two dictionary-scheme listings in Section 3.2.2 we can see that a big library of dictionary schemes can be built quite easily, and finding the optimum dictionary-scheme collections is thus not easy. Our current solution to this problem is heuristic. And systematic solutions are also possible. For example, we can make a dictionary-scheme collection by choosing its element dictionary schemes in a way similar to how Ensemble Selection (Caruana et al., 2004) selects classifiers; we can also do the selection by using common feature-filtering methods.

3.3 Predictor Builder

The predictor builder we use is quite simple. It uses the MultilayerPerceptron classifier from Weka (Hall et al., 2009) to do the learning and prediction. For Algebra 2008-2009, the number of learning iterations is set to 40. For Bridge to Algebra 2008-2009, it is set to 100. Other parameters just use the defaults. We haven't spent much time in trying out different predictor builders, so there could be much space for improvement here.

Table 1: Results of Our Submission and the Winner's Submission

Team	Algebra 2008-2009	Bridge to Algebra 2008-2009	Average
Us	0.282817	0.278134	0.280476
Winner	0.274568	0.271331	0.272952

3.4 Other Stuff

Our solution was implemented in Java. And besides the predictor builder, all other components were implemented using in-house code. On a computer with one Intel Xeon X3330 processor and 4GB of memory, our programs can finish processing (from original data to final result) either of the two challenge data sets in about five hours, consuming no more than 1.6GB of memory (32-bit JVM limitation, which caused lots of headaches to us). To be within memory budget, our programs had to write out lots of temporary files and pass the original data file multiple times, which slowed down processing considerably. According to our rough estimation, if given enough memory, the processing time can be reduced to less than 2 hours, using the same processor.

4. Result and Summary

Our result ranked sixth among all teams, and third among student teams. Table 1 gives our result, as well as the winner's. Submissions are measured by their root mean squared errors.

To summarize, our data mining strategy relies heavily on feature synthesis, using scoring machines as synthesizers. And instead of using the traditional data-mining procedure, we developed and used a new solution framework. The new framework can handle asymmetric training and test sets as well as non-atomic and variable-length attributes pretty well. The final-submission solution uses relatively simple components as building blocks, and many of its design decisions are made by crude heuristic processes, thus we think there could be much space for improvement in our work. We believe by combining our solution framework with a well designed, systematic solution generation method, better solutions can be obtained.

Appendix A.

A Score Scheme Using Normal Distribution: This is a relatively sophisticated score scheme. It only works with key scheme Anon Student Id+Step Id. Here is how it works: First, rank students according to their average Correct First Attempt. Students with higher averages get higher ranks. Second, use a normal distribution random number generator to generate a collection of numbers, the size of the collection should equal the number of students. Third, rank the random numbers to descending order. Fourth, take the Anon Student Id part of the key and look up the corresponding student's rank. Fifth, use the student's rank to look up the student's number, that is, the number having the same rank as the student. Sixth, take the Step Id part of the key and compute the corresponding step's average Correct First Attempt. And finally, multiply the student's number with the step's average Correct First Attempt. Return the result as the score. The rationale of this score scheme is to incorporate step difficulties and the assumption "students' performances are

normally distributed" into scores. With such info in them, the scores should have relatively strong relation to Correct First Attempt. And indeed this is the case. By using a single dictionary scheme Anon Student Id+Step Id:"this score scheme", we are able to get a result very close to 0.3, better than any other single dictionary scheme we have tried. Yet when the dictionary is added to a large collection like the one in Section 3.2.2, no improvement is gained. But since the evaluation experiment's coverage is far from complete, chances of this score scheme and its variations being useful are still high.

Other Splitters: There are other splitters that we have tried. One of them put the last two problems of every block into Dev-Test. Another put the last problem of every block into Dev-Test as usual, but let Dev-Training be the same as Training-Set. And there other still others, more or less similar to these two. It turns out none of these worked better than the one given in Section 3.1. One interesting thing to note is that if we use a splitter which simply let both Dev-Training and Dev-Test be the same as Training-Set, and add in some dictionary schemes which simply assign numeric ids to the keys, the solution will end up being just like a traditional data-mining procedure.

Split Multiple Times: This is an extension to the solution framework. Instead of splitting only once, we split multiple times. For example, we could first split Training-Set to Dev-Training and Dev-Test, and get Dev-Instances, and then split Dev-Training to Dev-Dev-Training and Dev-Dev-Test, and get Dev-Dev-Instances, and then use the union of Dev-Instances and Dev-Dev-Instances to train Predictor. We tried this once, but seen no improvements. Again, the experiments were crude, so this extension could actually be useful.

References

R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18. ACM, 2004.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.