Turning Datamining into a Management Science Tool: New Algorithms and Empirical Results[1]

Lee G. Cooper and Giovanni Giuffrida

The Anderson School, UCLA Computer Science Dept., UCLA

Abstract

This article develops and illustrates a new knowledge discovery algorithm tailored to the action requirements of management science applications. The challenge is to develop tactical planning forecasts at the SKU level. We use a traditional market-response model to extract information from continuous variables and use datamining techniques on the residuals to extract information from the many-valued nominal variables such as the manufacturer or merchandise category. This combination means that a more complete array of information can be used to develop tactical planning forecasts. The method is illustrated using records of the aggregate sales during promotion events conducted by a 95-store retail chain in a single trading area. In a longitudinal cross validation, the statistical forecast (PromoCast™) predicted the exact number of cases of merchandise needed in 49% of the promotion events and was within ± one case in 82% of the events. The dataminer developed rules from an independent sample of 1.6 million observations and applied these rules to almost 460,000 promotion events in the validation process. The dataminer had sufficient confidence to make recommendations on 46% of these forecasts. In 66% of those recommendations the dataminer indicated that the forecast should not be changed. In 96% of those promotion events where "no change" was recommended this was the correct "action" to take. Even including these "no change" recommendations, the dataminer decreased the case error by 9% across all promotion events in which rules applied.

---

[1] Correspondence should be addressed to: Lee Cooper, The Anderson School at UCLA, 110 Westwood Plaza, Suite B518, Los Angeles, CA 90095-1481. The authors wish to thank Dominque Hanssens, Donald Morrison, Wesley Chu and David F. Midgley for their helpful comments and the editors and referees of Management Science for their thoughtful critique and suggestions. We thank Sharon L. Bear, Ph.D. (BearWrite@aol.com) for her editorial assistance. This research has been supported by grants from Intel Corporation and software donations from Microsoft. The data were provided by *ems, inc*. The assistance of Penny Baron, Mike Swisher, and Bill Weissenberg is gratefully acknowledged.

Introduction

"Turning a mining tool loose in a large data set might produce more than 2,000 findings, all but 20 of them obvious, irrelevant or flawed . . . One tool told us income is higher for people who have big balances. Well, yippee," thus warns Mike Eichorst, Vice President of Predictive Modeling and Data Mining at The Chase Manhattan Bank Corp.'s Consumer Credit unit in New York in the article "Data Mining for Fool's Gold" (Computerworld, January 12, 1997). The growth of business databases has created the need. The rapid expansion of computer resources has created the potential. Utilizing the potential to fulfill the need has been hampered by a lack of communication between management scientists and computer scientists. This joint effort describes how datamining can augment traditional management science tools--market-response models in this instance (Blattberg and Neslin 1990, Lilien and Rangaswamy 1998, Rao and Steckel 1998)--and what we have learned from applying a new datamining algorithm to a large-scale, empirical effort aimed at tactical promotion planning.

Management science is action oriented. Businesses possess vast historical databases, and managers want to know how the information in them can help proscribe what actions to take in various sets of current and future circumstances. In our application, we already had a tactical forecasting tool (PromoCast™) that was calibrated to handle any of the over 150,000 SKUs for which a promotion event might be planned by a grocery retailer in a particular geographic market (Cooper et al., in press). That tool has to cope with the huge variability in results, from the six units that some well known brand might sell in one event to the 250,000 baskets of strawberries that suddenly appear on sale one February and move over the scanner. The statistical forecaster did this well. In the first pilot market, almost 49% of the forecasts predicted exactly the number of cases of product needed. Over 82% of the forecasts were within ± one case. However, Procter & Gamble might claim that, when Tide™ goes on sale at a large discount and appears in major ads, it gets a

bigger sales boost than estimated by the market-response model. A thousand other manufacturers could make a similar claim in each particular product category. Market-response models are not sufficiently robust to respond to the addition of 1,000 dummy variables for the manufacturers, 1,200 dummy variables for the merchandise divisions in a grocery store, 95 variables for the store-by-store effects, the possible interactions between these sets of indicators, or the possible interactions with the many other variables in the tactical forecasting model.

The statistical model has been designed to be transportable (after recalibration) across retailers and geographic markets. PromoCast™ uses 67 variables that capture how the history of each item (stock keeping unit or SKU) and the history of each store in a trading area combine with a proposed promotion plan to help retailers decide how much they should expect to sell in an upcoming promotion event. To characterize the promotion style the model uses unit price, the percentage discount, whether the promotion is an X-for-the-price-of-Y sale, main effects for ads and displays, two- and three-way interactions of ads, displays, and the percentage discount, and a large number of historical averages (e.g., the item's average promoted sales volume on similar promotions in the focal store).

In spite of tracking many influences, the parameters that reflect the importance of item-specific information in PromoCast™ may over-represent or under-represent the importance of that item's history for a particular manufacturer. Factors that are specific to a manufacturer, a retail store, or a geographic area do not fit well with the general scheme of a market-response model. A datamining algorithm, however, could be great at finding rules such as: "When manufacturer *A* underwrites a major promotion for its flagship brand *B* in major market *C*, the forecast tends to under predict by *D* cases." Nominal variables with many levels are obvious candidates from which to extract the local information that could improve a forecast. The statistical forecaster handles the quantitative (continuous) variables that tend to characterize all markets, while the dataminer handles the

nominal-scale variables that are more specific to a particular retailer and geography. We do this sequentially; first, the statistical forecast is developed on the quantitative variables, and then the dataminer is applied to the residuals from the statistical forecast. In other words, our datamining tool is oriented to discovering patterns in the residuals that correspond to local knowledge. We then use this local knowledge to form rules to improve the forecast. As we show, using these rules lets us know when we can be especially confident in the existing forecast, when we can expect a substantial overall reduction in forecasting error, and when we are not certain enough to act.

In this paper, we discuss our experience in designing and implementing a datamining tool that discovers patterns in the residuals that correspond to local knowledge. We do this on an enterprise scale. Our training database (used to develop rules) is a stratified random sample of more than 1.6 million records pulled out of a total database of more than 19 million records, reflecting retail grocery promotions from the 95 outlets of a major retailer in a large metropolitan area. Our population database represents about 30 months' worth of promotion events. Here we present a validity study based on out-of-sample results – a hold-out sample of almost 460,000 records that were collected after the statistical model was calibrated and after the local knowledge was mined.

The objective of the project was stated clearly at the beginning: how to produce forecasts that are useful for promotion planning. Grocers need to know how much stock to order for an upcoming promotion event. Grocers want to minimize inventory costs and out-of-stock conditions (often conflicting goals). Manufacturers want to maximize shipments, putting them somewhat at odds with the goals of the grocers. Possibly mitigating this conflict are the very large databases containing information on prior promotion experience for each separate SKU in each store within a retail chain for as far back as good records have been kept. Efficient Market Services, Inc. (***ems, inc.***) has been keeping such records on their clients' promotions. Databases exist for over 3,000

stores, and more are being developed.[2]

In Figure 1, the overall design is depicted. Cooper et al. (in press) developed a statistical forecaster called PromoCast™ that has a traditional market-response model orientation. It is a production forecast, not a custom model. Excluded from this model were nominal variables such as which manufacturer made the item to be promoted or what class of merchandise was being promoted (i.e., subcommodity). These two variables alone would add 2,200 dummy variables to the market-response model even before considering possible interactions of manufacturer or subcommodity with variables included in the model. A lot of information would be left in the residuals that would not easily be incorporated into a market-response model. This is the task we set up for the dataminer. We need a rule-induction algorithm to discover when the information in the excluded variables indicates that we should modify our forecast. Once a set of discovered rules is built, we can use such rules to *adjust* the forecast. This is the task of the "Corrective Action Generator" module. Such corrective actions suggest an offset (positive or negative) to be added to the forecasted value in order to get higher overall accuracy.

Insert Figure 1 about here.

Rule syntax and semantics

The datamining algorithm finds rules such as the following:

```
IF DCS = 'Gelatin' and TPR = 'Very High' and Mfr = General Foods'
THEN U_12_ = 0, U_4_11 = 58, U_3 = 221, U_2 = 1149, U_1 = 3583,
     Ok = 1115, O_1 = 7, O_2 = 1, O_3 = 0, O_4_11 = 0, O_12_ = 0
```

Where the independent variables in the "if conditions" have the following meaning:

- "DCS" stands for the triple "Department-Commodity-Subcommodity," identifying a particular class of merchandise being promoted (e.g., yogurts, gelatins, or prepared dinners).

[2] See www.emsinfo.com for more information.

- "TPR" identifies the level of the Temporary Price Reduction. Promotions usually involve some item price reduction. Values for this variable have been generalized to a set of five possible discrete values: *None*, *Low*, *Medium*, *High* and *VeryHigh*.
- "Mfr" identifies the manufacturer of the given product.

The other variables that we mined were:

- Promotion conditions "ME" identify a nine-fold, mutually exclusive and exhaustive classification of the ad and display conditions. Newspaper ads were classified as major ads, minor ads, or no ads. In-store displays were classified as major displays, minor displays, or no displays. The "ME" conditions were the cross-classification of this three-by-three classification.
- "Model" specifies one of the eight models used in PromoCast™. While the same variables (as described above) were used, separate parameters were estimated for each of the four major promotion-planning periods (one-, two-, three-, or four-week duration), crossed with slow-moving items versus fast-moving items. Slow-moving items were those that were expected to sell less than 10 units a week in an individual store (based on historic performance).
- "Store Node" allows for store-specific effects or interactions for each of the 95 stores belonging to one retail chain in the pilot market.

Errors in the forecast are expressed in a number of cases (i.e., the minimum order quantity for each particular SKU, usually 12 units in a case). For example, an error of –3 means that we underestimated the sales for that specific promotion by three cases ("U_3" class), while a value of 5 means that we overestimated five cases ("O_4_11" class). In our application, the entire set of possible errors has been generalized into a reduced set of 11 possible values for the class variable, namely:

```
O_12_:    Over by 12 or more cases;
O_4_11:   Over by 4 to 11 cases;
```

```
O_3:       Over by 3 cases;
O_2:       Over by 2 cases;
O_1:       Over by 1 case;
Ok:        No error
U_1:       Under by 1 case;
U_2:       Under by 2 cases;
U_3:       Under by 3 cases;
U_4_11:    Under by 4 to 11 cases;
U_12_:     Under by 12 or more cases;
```

The previous rule, for example, states a clear tendency to under forecast products in the subcommodity "gelatin" for the manufacturer "General Foods" when a large price discount is offered. As we will see later, we save a lot by specifying a corrective actions in such circumstances that simply increases our forecast by one case.

We turn now to a discussion of the datamining algorithm we call KDS (Knowledge Discovery using SQL – Structured Query Language) and the application of KDS to our problem.

Knowledge Discovery from Databases/Data Mining (KDD/DM)

Some remarkable industrial failures cooled down the initial enthusiasm of KDD/DM developers. The promised wonders of KDD/DM tools have too often resulted in some form of *obvious, superfluous* or *impractical* findings. Datamining advertisements portray a potbellied 30-ish man dressed only in diapers and tout such findings as "At 6:32 PM every Wednesday, Owen Bly buys diapers and beer. Do not judge Owen. Accommodate him." (Wall St. Journal, May 15, 1997, page B3). Such messages cause the eyes of management scientists, used to enterprise-scale applications, to glaze over. However, it is not our purpose to discuss the potential uses of datamining for mass customization of targeting, service, or customer support. Rather, we will demonstrate the ability of a datamining algorithm to find useful and well supported patterns in data that market-response models are not designed to harvest.

KDS is a highly scalable, rule-generating, datamining system that is not bound by physical memory, is bottom up, and requires little or no data preprocessing. KDS is implemented, following the tightly coupled model, with DB2®. The entire algorithm is executed as a sequence of complex Structured Query Language (SQL) queries sent to the database management system (DBMS). Each of these attributes is described below.

Rule Generation

The output of KDS is a set of *symbolic rules* in the form: "if *<pattern>* then *<class-distribution>*". The *pattern* is the conjunction of particular values for the independent variables (e.g., $a=A$ & $b=B$ & $c=C$, where A, B, and C are particular levels of the variables a, b, and c, respectively). At this time, KDS does not allow continuous variables; only discrete (nominal) variables currently can be part of the set of explanatory variables. This makes it an ideal complement for traditional market-response models that thrive on continuous variables, but have problems with large numbers of dummy variables.[3] The *class-distribution* is a frequency distribution of the dependent measure (number of case errors in our application) of all the input examples satisfying the condition specified on the "if" part. In the following, we refer to conjunctions of the form "$a=A$" as one-term patterns and conjunctions of the form "$a=A$ & $b=B$" as two-term patterns, and so on.

Top-Down versus Bottom-Up Algorithms

Most of the mining algorithms in the literature are based on a *separate-and-conquer* approach (Furkranz, 1996). In a nutshell, this is a recursive procedure where, at each recursion, the input dataset **I** is separated in two mutually exclusive and exhaustive parts $\mathbf{I}_1$ and $\mathbf{I}_2$ (say all "General

[3] Prior *ad-hoc* discretization can be used to transform continuous variables such as temporary price reduction (TPR) into levels of discount (e.g., None, Low, Medium, High and Very High).

Foods" promotions versus all other SKUs). The separation is performed in a way that maximizes a *function* ψ. Different algorithms use different functions; usually they tend to minimize the entropy of the class distribution on one of the subparts. Each separation generates a new rule R that covers all observations in $\mathbf{I}_1$ (say all "General Foods" promotions are under-forecast by two cases). In turn, after R is generated, $\mathbf{I}$ is assigned to $\mathbf{I}_2$ (the set of observations not covered by R – all SKUs that are not "General Foods" in this example) and the recursion continues on $\mathbf{I}$ (the *conquer* phase)[4]. The recursion halts as soon as no more splitting can take place (i.e., the database is smaller than a given threshold of support, namely the *minimum support*).

The most expensive part of these algorithms is the splitting phase in which some form of *"for each possible feature"* loop takes place in order to maximize ψ, that is, an exhaustive search is performed over the entire set of features. We argue that, when combinations of thousands of possible features have to be considered, this can be costly for large feature spaces. Besides the additional complexity of testing thousands of features, a *separate-and-conquer* approach may be inefficient since every possible combination of features is tested. Many of these combinations may not even exist in the input database (e.g., Yoplait diapers, Hamburger Helper batteries). This is a misplaced legacy inherited from machine-learning practice in which small feature spaces were the norm and the costs of testing features were small. When dealing with thousands of features, the possible combinations may be numerous, with many missing combinations. We refer to the process of testing all possible combinations of features as a *top-down* approach.

In contrast, KDS works in a *bottom-up* way, starting from the input database. Rules are built incrementally, starting from the simplest ones (one-term patterns) and then progressively proceeding

[4] Classification tree discovery algorithms (Quinlan, 1993) are based on a slightly different approach, namely *divide-and-conquer*. In such an approach, after the database is split into n subparts, the same procedure is recursively called on each subpart.

to more specialized rules (two-term, three-term, and so on). The first iteration generates all *observed* one-term patterns, the second generates all *observed* two-term patterns, and so on. Each iteration specializes all the patterns generated so far by adding a new term to the "if <*pattern*>". In each iteration, a (possibly large) set of new rules is added to the accumulating rule set. The iteration halts as soon as rules cannot be further specialized because their popularity (number of supporting records) drops below a specified *minimum-support* threshold. Since rule popularity decreases monotonically on each iteration, the process is guaranteed to terminate. The algorithm is sketched in the Appendix.

As stated above, in a *separate-and-conquer* approach, the first rule is induced from the entire input database. Then, all covered examples are removed and the second best rule is induced on what is leftover. Successive rules are thus induced from smaller and smaller portions of the database. Such progressive fragmentation of the input database yields reduced numerical support for rules discovered later in the induction process. Holte *et al.* (1989) demonstrate that a substantial proportion of the overall classification error is due to rules covering a small set of observations, which they call the "small disjuncts problem." Inducing rules on increasingly smaller sets (as done in *separate-and-conquer* algorithms) indirectly exacerbates the small disjuncts problem. KDS follows the *conquer-without-separating* strategy, proposed by Domingos (1996a and 1996b), that avoids the small disjuncts problem by discovering all rules from the entire input dataset.

No Memory-Bound Processing: Any form of discovery algorithm is inherently memory intensive. Most of the induction algorithms presented in the machine-learning literature exacerbate the small disjuncts problem by loading the entire data set (and the discovered knowledge) into the main memory. If the memory (physical and virtual) is full, the process stops. We have watched a C implementation of a standard algorithm, CN2 (Clark and Niblett, 1989), crash on a database of

about 70,000 records (with a large feature set) after about 10 hours of processing, even after allocating 400 Mbytes of main memory (physical plus virtual memory). Our management science applications are much larger than this. While recent developments of machine-learning techniques claim to reduce the cost of rule-finding algorithms (Domingos, 1996a), the cost is often computed under too ideal situations. Cohen (1995) tested the "Ripper" algorithm on a system with eight RISC processors and one gigabyte of physical memory. In that relatively ideal computational environment, the Ripper algorithm was the "best of class." Our need to be able to scale management science applications to the enterprise level, however, implies that, in many contexts, not enough physical memory will be available. The virtual memory facility will be needed. The cost of swapping between physical and virtual memory invalidates the original cost estimates for an algorithm.

We tried Ripper on our training database using a Windows NT dual processor system (2 × 200Mhz Pentium Pro) equipped with 128 Mbytes internal memory and enough virtual memory to avoid crashing the algorithm. Even with no other tasks running at the same time, Ripper executed for 21 days without finishing. Datamining algorithms based on physical and/or virtual memory are not practical for management science applications of this scope. KDS, in contrast, is implemented in a tightly coupled (cf. Agrawal and Shim, 1995b and 1996a), client-server model described below. Whenever the problem size is too large to fit into physical memory, the kind of client-server model described below should have a substantial practical advantage over CN2 or Ripper.

Minimal data preprocessing: Most discovery algorithms require the input data to be in a specific format, usually a single, flat-text file. This requires an *export* operation from the DBMS hosting the data (i.e., the mining tool is *decoupled* from the DBMS). Exporting a very large database can be a lengthy and tedious process, causing an extremely large text file to be generated. The benefits of the relational data model can no longer be exploited. This leads to data replication and redundancy that

can make the flat file much larger than the size of the original relational database. Furthermore, data need to be clean and formatted as requested by the mining tool. Data preprocessing can easily count for the 70-80% of the total KDD processing time. In the literature, very often, algorithms are compared on computation time (efficiency) without considering the time spent in data preprocessing.

In a *loosely* coupled, client-server model (Agrawal and Shim, 1995b and 1996a) the mining tool extracts the records from the DBMS one at a time. Such an operation is typically performed through exploitation of cursors in an embedded SQL application. While this approach eliminates the hassles of generating and handling large flat text files, in loosely coupled models, substantial data communication takes place between the client (the mining tool) and the server. The entire database has to be transferred, record by record, since the processing is performed entirely on the client side while the data reside in the DBMS.

KDS is implemented as a *tightly coupled*, client-server model, in which the largest part of the mining process is implemented on the server. The communication traffic between the two systems is reduced to delivery of commands and retrieval of results. The client acts as a control to synchronize the different phases of the process. The complete task is achieved by a sequence of complex queries execution and/or *calls* to User Defined Functions (UDFs) (cf. Agrawal and Shim, 1995b). We believe that the tightly coupled, client-server model is by far the most promising for developing highly scalable, datamining processes, for reasons relating to how rules are generated, organized, and ranked. These issues are discussed below.

Rule generation and organization: In most induction algorithms, the rule generation and rule ranking phases are tightly integrated. A rule-scoring mechanism generates the best rule for each iteration. This is fine for exploration, but not for action-oriented managerial applications. In

management science applications, we need to see what stored knowledge tells us about a current situation and act accordingly. We may not need to take action until long after the learning. However, we may need to increment what we learn with new information. The advantage, then, goes to a method that separates the learning phase from the action phase. In KDS, there is a crisp separation between the rule generation (learning) phase and the action (rule ranking and selection) phase. KDS creates all the rules from the input database and arranges them in a *rule network*. The rule ranking and selection task is postponed until the action phase (discussed below). KDS typically generates a large set of discovered rules. The *rule network* optimizes rule retrieval and speeds up the classification task (i.e., the task of finding which rules in the network apply to a given new situation). An example of *rule network* is seen in Figure 2. The lowest levels of the rule network contain the one-term patterns. Up one level are the two-term patterns, and so on. This architecture simplifies the process of selecting all rules containing a specific pattern. They are simply identified by all the ancestors of the node containing the pattern of interest. Each node of the rule network contains the specification of the pattern itself and the class distribution vector.[5]

Insert Figure 2 about here.

The Action Phase – Classifying and Acting on New Examples

Once the rule network has been created, we move to the *action phase,* in which classification of new examples takes place. With a rule network in place, a new event occurs. In our case, the new event is a planned promotion for which a forecast has been made. We need to determine whether to alter that forecast, given the local knowledge in the rule network. We must figure out which rules apply to

[5] In the actual implementation of KDS, we also record entropy, rule coverage, and number of features. This additional information is useful in speeding up the classification process.

this new event and take the appropriate action. Classification in KDS is performed through the following steps:

1. **Rule selection**: find all the rules covering the example to be classified.

2. **Rule ranking**: select the best rule(s) according to the ranking criterion.

3. **Example classification**: assign the most likely class of the chosen rule(s) to the input example (e.g., is it most likely that this event is over-forecast by two cases?).

As explained above, the rule network makes the selection of all rules applying to a new case efficient. For instance, given the rule network of Figure 2 and the input example *{ME = 'Major Ad and No Display,' Mfr = 'General Foods,' Tpr = 'None,' DCS = 'Yogurt'}*, Figure 3 shows all the rules in the rule network that are triggered by the input example (i.e., the set of rules covering the given example). Notice that rules whose pattern contains the feature *DCS = 'gelatin'* have not been activated since such a condition does not occur in the input example. The selection algorithm starts from the bottom of the rule network by activating the one-term rules corresponding to the features of the input example. Then the activation is propagated upward, and each higher node is activated if all its children are active. The activation goes up to the highest nodes of the network. At this point, all rules covering the input examples are marked. Among these rules a ranking has to be performed that indicates to which action class this new example most likely belongs.

Insert Figure 3 about here.

Entropy-based rule ranking is widely exploited in rule classifiers. Entropy is computed from the class distribution vector. Because there can different costs associated with certain types of misclassifications in the class distribution, we felt that a simplified procedure would be more robust (cf. Hand, 1997, p. 7). In the *winning-group* procedure we developed, the class distribution is rearranged to perform only three types of corrective actions: No-Action, Add Cases, and Subtract

Cases. Basically, we use the most populated class in the original distribution to decide the *winning group*. For instance, let us consider that the most populated class is "Under_3." As such, the *winning group* will be a new class whose population is obtained by summing up all the Under classes together, while the *others* is obtained by summing up all the "Over" classes plus the "Ok" class. If "Ok" is the *winning group,* the *others* is obtained by summing up all the "Over" classes plus the "Under" class. The entropy is then computed on this two-class distribution. We then store the rule along with this normalized entropy. An algorithm analysis is presented in the Appendix.

The Action Phase – Corrective Actions

The rule network assures us that we can easily find the set of rules that apply to a new event. We still, however, must decide which corrective action to take. To aid this, each rule is annotated with the *entropy* and *confidence* value. The entropy value is computed by the formula: $E = -\Sigma\, p\ log(p)$, where the estimates of the p values come from the relative frequencies in the class distribution for a rule in the calibration data set. The *confidence* value is computed as: $10{,}000 \times (1 - E')$, where E' is the entropy normalized to remove differences due only to the number of categories in a class distribution. *Confidence*, basically, gives us an estimate of how *strong* the rule is, that is, how much we *trust* the rule. A class distribution with only one non-empty class, out of the 11 possible classes, gives us the highest *confidence* value. Conversely, a uniform distribution, in which all the classes are equally populated, gives us the highest *entropy* and the lowest *confidence*.

A *corrective action* is taken following a forecast in order to try to reduce error. Corrective actions are suggested by the induced rules. Different types of corrective actions can be taken. The simplest one is based on adjusting the forecast value according to the most likely class. Consider again the rule:

IF DCS = 'Gelatin' and TPR = 'Very High' and Mfr = General Foods'

THEN U_12_ = 0, U_4_11 = 58, U_3 = 221, U_2 = 1149, U_1 = 3583, Ok = 1115, O_1 = 7, O_2 = 1, O_3 = 0, O_4_11 = 0, O_12_ = 0

Here, the simplest (most intuitive) corrective action would suggest incrementing the forecasted value by one, since the most populated (most likely) class is U_1 (indicating that our forecaster tends to *under-forecast* under these specified conditions). This simple action leads to overall improved accuracy. That is, without the corrective action the *total case error* for the previous rule is computed by the class distribution as:

*12*0+4*58+3*221+2*1149+1*3583+0*1115+1*7+2*1+3*0+4*0+12*0=6785*

Once we perform the corrective action of adding one case to all estimates, we get the following total case errors:

*12*0+4*0+3*58+2*221+1*1149+0*3583+1*1115+2*7+3*1+4*0+12*0=2897*

By shifting one case up, we basically fix the 3,583 *U_1* cases (that now lead to an "0" error); we also reduce the error for all *U_xx* cases. At the same time, we increase the errors for *Ok* (the 1,115 *Ok* cases now have an error of one case each) and all the *O_xx* classes. However, the frequency distribution after the correction yields a substantial error reduction [*(6785-2897)/6785 = 57.3%*] for the set of examples covered by that rule. Intuitively, no corrective action should be taken when the most populated class is "Ok."

While for this example almost 77% of the forecasts were within ± one case, across all events in this pilot market, over 82% of the forecast errors are within ± one case. In light of this *a priori* knowledge, we restricted our actions to a maximum of ± one case. We used this method in the results below, but hope to generalize the method in the future.

Results

The parameters of the statistical forecast were calibrated on 1.3 million observations from a

stratified random sample of promotion events from the prior 30 months in a large metropolitan market area for 95 stores of a retail chain. The dataminer was run on nonoverlapping 1.6 million observations from the same event population, and 28,187 rules were generated. The summaries reported here are based on a large dataset (459,526 records) from a hold-out, cross validation period that occurred months after the parameters of the market-response model had been estimated. The "Ok" class is by far the most populated. As stated earlier, for almost 49% of the promotion events, the market-response model forecast the correct number of cases. For 82% of the events the model was within ± one case and for 90% of the events the model was within ± two cases. The average absolute error is far less than one case per promotion event. This gives a clear idea of how well the statistical forecaster works, even before applying the dataminer. The task of the dataminer is therefore extremely challenging. Even a small improvement (in terms of error reduction) is hard to achieve since it is on top of an already highly accurate system.

The *confidence* for acting on a rule was set relatively low (900 out of a maximum value of 10,000)[6]. The dataminer had sufficient *confidence* to recommend action on 46% of the forecasts (209,912 events). In the spirit of the physician's rule to first do no harm, the dataminer recommended "No Change" in 66% (138,614) of these events. "No change" was the correct "action" to take 96% of the times it was recommended. In such instances the dataminer adds credibility to the original forecast.

The dataminer compensates for the kind of patterns no manager would be expected to recall and no market-response model would traditionally incorporate. For example, for 2,635 four-week promotion events for cat food, with a medium level of price reduction, the PromoCast™ over-

[6] The minimum confidence threshold value depends upon the current user's goal. The higher the value, the fewer rules are created and the fewer records are classified. However, these fewer records are classified with a higher degree of confidence. More applied studies have to be conducted before precise guidelines can be developed.

forecasts 58% of the time and under-forecasts 26% of the time. Simple corrective action leads to a 46% reduction in case errors. For 350 21-day promotion events for Pedigree Dog Food (with a medium level TPR), PromoCast™ under-forecasts 88% of the time and over-forecasts just 2% of the time. Simple corrective action reduces case errors by 79%. For long (28-day) promotions for Yoplait Yogurt (2,445 events), PromoCast™ over-forecasts 59% of the events and under-forecasts 24%. Simple corrective action reduces case errors by 42%. For short (seven-day) promotions for Dannon Yogurt (1740 events), PromoCast™ under-forecasts 78% of the events, while over-forecasting just 3%. Simple corrective action reduces case errors by 60%.

In this validation study, the benefit showed mostly on the under-forecast side. That is, the dataminer tended to catch somewhat more situations where the statistical model under-forecasts sales. This may, in part, be due to the truncation that occurs in out-of-stock conditions. If the store runs out of stock the forecast may appear to be too large for a reason that the dataminer cannot detect. To a minor extent, the corrective actions worsened the over-forecast classes. Of course in the instances when the dataminer suggested that we "correct" an already accurate forecast, the dataminer worsened case errors. For this to be managerially acceptable we need the overall effect to be beneficial, which it is in this case. The cumulative case error from PromoCast™ for the 209,912 events in which rules applied was 112,860 cases of merchandise. Across all actions taken (including "No Change"), the dataminer reduced errors by 10,117 cases (8.9%). To put the 8.9% across the board improvement in perspective, we report the efforts of Krycha (1999). He provided two teams with the data used by PromoCast™ and KDS for the pilot market (1.2 million records). One team consisted of graduate students and two consultants from the SAS Institute Austria. They used the SAS Enterprise Miner™ to try to reduce case errors. The other team consisted of graduate students and two consultants from Eudaptics (a statistical consulting group in Vienna that specializes in self-organizing maps). This group used SOMine™ to try to reduce case errors. After a semester of

effort, both groups reported that they could not improve on PromoCast™. Viewed from this perspective, even the 8.9% across the board improvement seems more impressive.

Table 1 summarizes the rules that were used to change forecasts for 209,912 events. Table 2 summarizes the rules that were used to support not changing the forecasts for 249,614 events. We mined up to four-term rules. Of the rules we used, approximately 75% were either two-term or three-term rules. The relative frequencies for the number of terms in a rule (i.e., the bottom row of each table) were stable between rules pointing to a change and rules indicating no change. Over 85% of the activated rules had more than one term. Remembering back on the problem of adding 1,200 dummy variables for merchandise divisions and 1,000 dummy variables for manufacturer, we now see that these additions grossly underestimate the specification problem. Over 85% of the actions we take invoke rules reflecting higher-order interactions. The dataminer represents an enterprise-scale method for finding these interactions.

Insert Tables 1 and 2 about here.

The cell percent reflects what percentage of n-term rules used the variable in that particular row. Some interesting patterns emerge. Note in Table 1 (change rules) that no one-term rules appear for Promotion Condition - ME, Store Node or TPR. For Promotion Condition – ME, this is not too surprising since a similar term already appears in the model, leaving only higher-order interactions potentially unused. We also would not expect to have to change all the forecasts relating to a particular Store Node. TPR is a five-step, categorical variable that is monotonically related to the discount variable in the PromoCast™ model. However, we would be mistaken to assume that all the information in TPR is used in the market-response model. We see this when we compare the 0% of events that invoked one-term rules used TPR to change forecasts (cf. Table 1), while 20% of events that invoked one-term rules used TPR indicate no change in the forecasts (cf. Table 2). Further investigation shows that almost all of these "no change" rules involved lower levels of TPR –

probably reflecting low sales for these event for which one case was sufficient. Similarly, 42% of one-term rules in Table 2 involve Model. Further investigation shows that when we forecast for slow movers for longer events (two-, three- or four-week events), we can have extra confidence in the original PromoCast™ forecast.

Overall, 30% of the events invoked rules involving manufacturers or merchandise categories (or both). To get a better feel for how the dataminer would help a manager, we will look at these rules for the biggest manufacturers and the biggest merchandise categories. Table 3 summarizes the datamining results for the eight most frequently promoted categories (sub-commodities). Carbonated beverages are difficult to predict. Where KDS rules apply, the PromoCast™ errors are nearly four times as big as the average category. KDS reduces these errors by 14.3%. Even bigger percentage error reduction occurs for prepared meals, yogurts, and ice creams. For cookies, crackers and savory snacks, and shampoos the error reduction is modest. Notice that these categories have small average errors. The rules that KDS finds for the most part say "No Change."

Table 4 shows the datamining results for the eight most frequently promoted manufacturers. By far, the largest is the private label category in which the retailer is presented as if it were the manufacturer. This "manufacturer" cuts across so many areas that we should not be surprised that it reflects just about the average error reduction of 8.9%. Double-digit error reductions occur with dataminer rules for General Mills, Kraft, Coca Cola, Frito Lay, and General Foods. Rules for Procter & Gamble give a 4% error reduction even though the PromoCast™ forecasts for Procter & Gamble events in these instances are much more accurate than the average.

Insert Tables 3 and 4 about here.

Discussion

Using the discovered rules, we can spot sub-domains in which the PromoCast™ forecast performs either brilliantly or poorly. The symbolic rule representation gives us a precise, understandable

description of these sub-domains. There are two primary ways to take advantage of such information. The first way would be to review the statistical forecast model itself in order to embed such information. This, in essence, would correct for misspecification of the original statistical model. This is what modelers typically do. They find what is missing from the original specification and modify accordingly. However, since any particular rule typically covers only a small percentage of the total event pool, and since the data driving the improved performance due to the dataminer are typically nominal-scale variables, the potential for directly modifying the statistical forecast model is small. We are trying to achieve a synthesis of methods. PromoCast™ is designed to be transported across markets and retailers. The 67 variables in the basic model will have different importance in different applications, but the totality (explained variance) should be relatively stable.[7] The customization to each retailer-market combination involves the development of local knowledge. Here the marketing value associated with such information may be large. Manufacturers want forecasts tailored to their individual merchandize lines. Category managers need help in handling such demands. The dataminer essentially provides that kind of mass customization. While the results presented here are for a cross validation dataset, the dataset on which the knowledge is developed could be used to inform managers when they can have extra confidence in the original PromoCast™ forecast (i.e., when the dataminer indicates the forecast is "Ok," when local knowledge can be used to improve the forecast, and when we are uncertain, that is, when there are no rules covering a forecast).

Future directions

Many issues are still open to investigation. Our highest priority concerns the incremental acquisition of knowledge. Induced knowledge should be persistent and updateable over time in a data-intensive,

[7] Applications of PromoCast™ in five other pilot markets support this. However, the KDS algorithm has only been applied in the pilot market described in this report.

dynamic environment. Incremental learning has received some attention in recent years (Agrawal and Psaila, 1995a, Shan and Ziarko, 1995, Thomas *et al.*, 1997), but most of the machine-learning-rooted, rule-induction algorithms are based on a "all-at-once" execution model. This means that data are read and the rules are generated in just one step. Updating the discovered knowledge with these algorithms requires a fresh re-mining of the entire database. This situation may be unacceptable in a dynamic, data-intensive environment. Think of a grocery retailer whose cash registers process purchases of thousands of different items daily. A good KDD/DM system should be able to update the knowledge discovered so far with the new incoming records. Because of the separation between the rule-generation phase and the rule-ranking phase, KDS is capable of doing this. This capacity has yet to be tested in real applications.

Another new area involves a reorganization of the rule network to extend the rule syntax by allowing set-valued features in the rule antecedents (Cohen, 1996). This is like considering the "or" rules as well as the "and" rules emphasized so far. Consider the following two rules:

```
IF  Tpr= None &                        IF  Tpr= None &
    Mfr_Code=General Foods &               Mfr_Code= General Foods &
    DCS= Luncheon Meats                    DCS=Puddings
THEN No = 0, Ok = 115                  THEN No = 0, Ok = 113
```

They can be merged into the following single rule:

```
IF Tpr= None & Mfr_Code= General Foods & DCS in {Luncheon Meats, Puddings}
THEN No = 0, Ok = 228
```

The variable DCS has been combined into a set of values; notice also the combined class distribution. The merging of such rules can take place thanks to the overlapping nature of the antecedents and the uniform class distribution of the two rules. This extension would draw the dataminer closer to the domains in which CART algorithms are used (Brieman et al. 1984). CART applied to nominal-scale variables such as those used here, looks at all possible binary splits. This is totally impractical for a variable such as manufacturer with 1,000 levels or DCS with 1,200 levels.

We also have begun investigation of a hierarchical or sequential approach to the action stage of datamining for management science applications. We would first decide whether we should change the forecast at all. We then would decide whether this new event is going to be an over-forecast or an under-forecast. If an under-forecast, we would then try to decide how many cases under. Such an approach would allow us to probe more specifically into what contributes to over- or under-forecasts. More importantly, this approach should allow us to extend the range of corrective actions beyond the simple ± one case described here. Preliminary research has been encouraging, but more development is needed.

Limitations

Any technique that focuses on using history (stored knowledge) to help correct future actions has inherent limitations in new product research and forecasting. Neither PromoCast™ nor KDS have anything to say about forecasts for new products. In both of these applications, historical data are the strategic asset being exploited.

Out-of-stock conditions also create a limitation for KDS. Some of the errors arise when the forecast would be accurate if only the store did not run out of inventory. So we are more likely to observe errors associated with overstocking than with understocking. The truncation of errors associated with this issue is very difficult to handle. Cooper et al. (in press) discuss some aspects of the issue, but a full treatment is not possible within either the statistical model or the dataminer.

The other obvious limitation deals with uses of KDS for finding model misspecification in PromoCast™. Our design is one that focuses on using KDS on information that is not easily incorporated into a traditional market-response model. To now turn around and say that we could use KDS to find variables that could be included in the specification of PromoCast™ is somewhat awkward. KDS is best used in the discrete-variable space, PromoCast™ in the continuous variable space. There are, however, many exceptions. PromoCast™ uses indicator variables for ads and

displays, and holiday effects. KDS breaks the continuous variable TPR into bins for "Very High TPR" and the like. As such, it is possible to use KDS on such "binned" variables as a way to look more systematically at model residuals (for possible misspecification). What we really advocate, however, is that researchers use any tool available to study the residuals from their models. Learning from what is left behind in model specification is a fundamentally important part of model building. KDS does this naturally when what is excluded is of a different data type (discrete data) than that used in the base model (continuous data).

Conclusions

The sequential application of statistical forecaster and dataminer provides a natural way to use a broader set of information that easily can be used by either. Sure, it is theoretically possible to use market-response models to incorporate the 28,000 rules we found in this pilot market. Sure, it is theoretically possible to discretize all the variables used in the market-response model so that they could be analyzed with the dataminer. But we feel strongly that we are better off using each of the techniques where each best fits. We feel the benefits demonstrated so far justify our continued exploration of these techniques.

References

Agrawal, R. and G. Psaila. 1995a. Active Data Mining. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*. AAAI Press, Montreal.

Agrawal, R. and K. Shim. 1995b. Developing Tightly-Coupled Data Mining Applications on a Relational Database System: Methodology and Experience. IBM Research Report RJ 10005(89094).

Agrawal, R. and K. Shim. 1996a. Developing Tightly-Coupled Data Mining Applications on a Relational Database System. In *Proceedings of the Second International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Portland, OR.

Agrawal, R., M. Metha, J. Shafer, and R. Srikant, 1996b. The Quest Data Mining System. In *Proceedings of the Second International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Portland, OR.

Anand, T. 1996. The Process of Knowledge Discovery in Databases: A Human-Centered Approach. In *Advances in Knowledge Discovery and Data Mining*. AAAI Press / The MIT Press.

Blattberg, R. C. and S. A. Neslin. 1990. *Sales Promotion: Concepts, Methods, and Strategies.* Prentice Hall, Englewood Cliffs.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*, Wadsworth International Group, Belmont, CA.

Cheung, D. W. and J. Han. 1996. Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. In *Proceedings of the 12th ICDE*. New Orleans, LA.

Clark, P. and T. Niblett. 1989. The CN2 Induction Algorithm. *Machine Learning* **3** 1 261-283.

Cohen, W. W. 1995. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*. Lake Tahoe, CA.

Cohen, W. W. 1996. Learning Trees and Rules with Set-valued Features. *Proceedings of the Eighth Annual Conference on Innovative Applications of Artificial Intelligence, AAAI-96*. Portland, Oregon.

Cooper, L. G., P. Baron, W. Levy, M. Swisher, and P. Gogos. (In press). PromoCast™: A New Forecasting Method for Promotion Planning. *Marketing Science*.

Domingos, P. 1996a. Linear-Time Rule Induction. In *Proceedings of the Second International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Portland, OR.

Domingos, P. 1996b. Unifying Instance-Based and Rule-Based Induction. *Machine Learning* **24** 141-168.

Fayyad, U. M., G. Piatetsky-Shapiro, P. Smyth, and R. Brachman. 1996. From Data Mining to Knowledge Discovery: An Overview. In *Advances in Knowledge Discovery and Data Mining*. AAAI Press / The MIT Press.

Feldman, R., Y. Aumann, A. Amir, and H. Mannila. 1997. Efficient Algorithms for Discovering Frequent Sets in Incremental Databases. In *Proceedings of the 1997 SIGMOD Workshop on DKMD*. Tucson, AR.

Furkranz, J. 1996. Separate-and-Conquer Rule Learning. Technical Report OEFAI-TR-96-25. Austrian Research Institute for Artificial Intelligence.

Hand, D.J. 1997. *Construction and Assessment of Classification Rules.* John Wiley & Sons, Chichester.

Holte, R. C., L. E. Acker, and B. W. Porter. 1989. Concept Learning and the Problem of Small Disjuncts. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, Detroit, MI.

John, G. H. and B. Lent. 1997. SIPping from the Data Firehose. In *Proceedings of the Third International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Newport Beach, CA.

Kakemoto, Y. and S. Ohsuga. 1997. KDD Process Planning. In *Proceedings of the Third International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Newport Beach, CA.

Krycha, K. A. 1999. Case Study Growmart: The Effects of Promotional Activities on Sales. Übung aus Verfahren der Marktforschung Endbericht. Universität Wien, Institute für Betriebswirtschaftslehre.

Lilien, G. L. and A. Rangaswamy. 1998. *Marketing Engineering: Computer-Assisted Marketing*

Analysis and Planning. Addison-Wesley, Reading, MA.

Liu, B., W. Hsu, and S. Chen. 1997. Using General Impressions to Analyze Discovered Classification Rules. In *Proceedings of the Third International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Newport Beach, CA.

Ng, V. T. and C. Y. Wong. 1996. Maintainance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. In *Proceedings of the 12th ICDE*. New Orleans, LA.

Piatetsky-Shapiro, G., R. Brachman, T. Khabaza, W. Kloesgen, and E. Simoudis. 1996. An Overview of Issues in Developing Industrial Data Mining and Knowledge Discovery Applications. In *Proceedings of the Second International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Portland, OR.

Provost, F. J. and V. Kolluri. 1997. A Survey of Methods for Scaling Up Inductive Learning Algorithms. Submitted to *Data Mining and Knowledge Discovery Journal: Special Issue on Scalable High-Performance Computing for KDD.*

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Rao, V. R. and J. H. Steckel. 1998. *Analysis for Strategic Marketing.* Addison-Wesley, Reading, MA.

Shan, N. and W. Ziarko. 1995. Data-Based Acquisition And Incremental Modification Of Classification Rules. *Journal of Computational Intelligence* **11** 2 357-370.

Silberschatz, A. and A. Tuzhilin. 1996. What Makes Patterns Interesting In Knowledge Discovery Systems. *IEEE Transactions On Knowledge Discovery and Data Engineering.* **8** 6 970-974.

Thomas, S., S. Bodagala, K. Alsabti, and S. Ranka. 1997. An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. In *Proceedings of the Third International Conference on Knowledge Discovery & Data Mining*. AAAI Press, Newport Beach, CA.

Zhong, N. and C. Liu. 1997. KDD Process Planning. In *Proceedings of the Third International*

Conference on Knowledge Discovery & Data Mining. AAAI Press, Newport Beach, CA.

Appendix – Algorithm Description and Analysis

Insert Exhibit 1 about here.

KDS works by a progressive rule specialization. The nth iteration creates all the n-term rules existing in the input database. Rule coverage (i.e., the number of records "covered" by a rule) must decrease monotonically at each iteration. The process is halted as soon as further specialization leads to coverage below the specified *minimum support* for all new generated rules. Only observed combinations of features are considered when building rules, which is much more efficient than algorithms that process all theoretical combinations of features (Clark and Niblett 1989, Cohen 1995). ***R[N]*** represents the set of N-term rules. The set **S** contains all the N-term rule combinations assigned to the current record. For instance, say the input record is: *{a=10, b=low, c=john}*, then the set **S** at the second iteration (N=2) contains all two-term conjunctions: *{a=10* & *b=low, a=10* & *c=john, b=low* & *c=john}*. Likewise, the set **T** is constructed from the elements of **S**. For instance, for the element *{a=10, b=low}* of **S**, **T** would be: *{{a=10}, {b=low}}*, a set of (N-1)-term patterns. The notation ***R[N].supp(X)*** specifies the popularity of the pattern X in the rule set **R**[N]. *X.class* is the class value of the input example *X*, while ***R[N].class (Y, C)*** is the frequency of the class *C* for the rule *Y* in the rule set ***R[N].***

The rule generation performs a total of k iterations, where k is either the maximum number of terms in the patterns before the coverage drops below the *minimum support* value (for all the new rules), or the maximum number of terms in the rule antecedents that we feel able to interpret. The upper bound for k is the number of independent variables. Therefore, the "while loop" in the algorithm has a cost that is linear in k and e, where e is the number of input examples. The nth

iteration exploits the results of the (n-1)th iteration. For instance, to add the new pattern "a&b&c" at the 3rd iteration, it is necessary (but not sufficient) that "a&b", "a&c", "b&c" are all previously supported. The size *s* of the set **S** in the algorithm (shown in Exhibit 1 at nth iteration) is *a!/[n!(a-n)!]*, where *a* is the total number of independent variables. The set **S** contains the candidates for new patterns to be added to the rule set. For each element of **S** the set of subpatterns is generated and stored in the set **T** whose size we refer as *t*. For each element of **T** a lookup (with logarithmic cost) is executed until one element is not supported or all the elements have been verified to be supported. In the worst case *t* lookups have to be performed for each of the *s* elements of **S**. The total cost becomes linear in *k, e, s, t* and *log(l)* where *l* is the size of the **R**[n-1] set at the nth iteration. Furthermore, for each iteration a pruning loop is executed to remove all new rules that are not supported (i.e., had fewer instances than the user definable, minimum-support threshold). This component has a minimal cost that can be omitted in the cost computation. In the previous computation *s* is a function of the number of combinations of independent variables, so the cost of KDS rule generation phase increases roughly with the square of the size of the variable space. The total cost is not a function of the number of features (i.e., number of levels of a nominally valued independent variable). This makes KDS more suitable for databases with a large number of records, a small number of independent variables each of which has a large number of levels or features. The cost independence from the number of features makes KDS *noise tolerant*. Noise in databases results in some features with minimal support. The *bottom-up* induction style of KDS leads to very little additional work for infrequently supported features (recall that no *"for each possible feature"* loop takes place in the algorithm). Furthermore, poorly supported features are promptly dropped by the pruning loop at the end of each iteration. Noise represents a difficult issue for many induction algorithms whose cost increases an order of magnitude in presence of noisy data. Some algorithm's

performance worsens to being a quartic function (e^4) in noisy domains, where e is the number of training examples (Cohen, 1995).

In KDS rule ranking occurs during the classification of new examples. Rule selection is actually executed prior the ranking, only rules applying to the example to be classified are selected. This greatly reduces the rule search space for the rule-ranking activity. The rule-selection algorithm described above has a small cost, that is the cost of looking up each feature of the input example in the *one-term rule set*. Then an upward search of the rule network will mark all parent rules. This last operation has negligible cost close due to the indexed structure of the pattern network. This leads to a total cost for each new record to be classified being a linear function of v and $log(l)$ where v is the number of independent variables and l is the total number of one-term patterns in the rule network.

The execution of KDS on our large database took a total of five hours with a tightly coupled implementation with DB2® in a Windows NT system. This is a substantial improvement compared to the 21 days (and still counting) for the decoupled Ripper implementation.

Table 1. Events Where Rules Change Forecast, N=209912.

	Terms in Rule			
Variable	**1**	**2**	**3**	**4**
Promotion Condition - ME	0%	9%	17%	22%
Store Node	0%	10%	11%	7%
Model	27%	29%	27%	23%
Manufacturer	34%	17%	11%	12%
Subcommodity - DCS	39%	19%	11%	13%
TPR	0%	16%	22%	22%
Pct. of Rules in Column	14%	34%	39%	13%

Table 2. Events Where Rules Do Not Change Forecast, N=249614.

	Terms in Rule			
Variable	**1**	**2**	**3**	**4**
Promotion Condition - ME	2%	15%	21%	22%
Store Node	3%	16%	14%	8%
Model	42%	20%	21%	22%
Manufacturer	13%	12%	11%	15%
Subcommodity - DCS	19%	14%	11%	12%
TPR	20%	23%	23%	21%
Pct. of Rules in Column	13%	37%	38%	12%

Table 3. Datamining Results for the 10 Most Frequently Promoted Categories.

Subcommodity	No. of Events covered by rules	PromoCast Errors	PromoCast + KDS Case Errors	Case Improvement	Percent Improvement
Carbonated Beverages	6827	11165	9573	1592	14.3%
Cookies	4440	1651	1632	19	1.2%
Prepared Meals	4009	2265	1892	373	16.5%
Frozen Pizza	2460	1034	983	51	4.9%
Yogurts	3737	4445	3214	1231	27.7%
Ice Creams	5259	3827	3223	604	15.8%
Crackers & Savory Snacks	3483	1198	1182	16	1.3%
Shampoos	5503	1087	1087	0	0.0%

Table 4. Datamining Results for the 10 Most Frequently Promoted Manufacturers.

Manufacturer	No. of Events covered by rules	PromoCast Errors	PromoCast + KDS Case Errors	Case Improvement	Percent Improvement
Store Private Labels	29972	21983	20028	1955	8.9%
Procter & Gamble	7179	2570	2465	105	4.1%
Nabisco	3805	1466	1435	31	2.1%
General Mills	3745	2808	2231	577	20.5%
Kraft	3663	3081	2761	320	10.4%
Coca Cola	1797	4086	3536	550	13.5%
Frito Lay	616	898	774	124	13.8%
General Foods	3104	3059	2452	607	19.8%

Exhibit 1

```
I = input database;
N = 1;
Flag = True;
While Flag
   Flag = False;
   R[ N] = {};
   For each record W in I do
      S = {N-term patterns from W};
      For each X in S do
         T = {(N-1)-term patterns from X};
         If (N=1) or
           (all elements in T are supported)
then
            Flag = True;
            If X ∈ R[ N] then
               R[N].supp(X) = R[N].supp(X) + 1;
            Else
               R[N] = R[N] ∪ {X};
               R[N].supp(X) = 1;
            End If
            Increment R[N].class(X, X.class);
         End If
   End For

   ; Pruning by minimum support
   For each Y in R[N] do
      If R[N].supp( Y) < min-supp then
         R[N] = R[N] – Y;
      End If
   End For
   N = N + 1;
End While
```

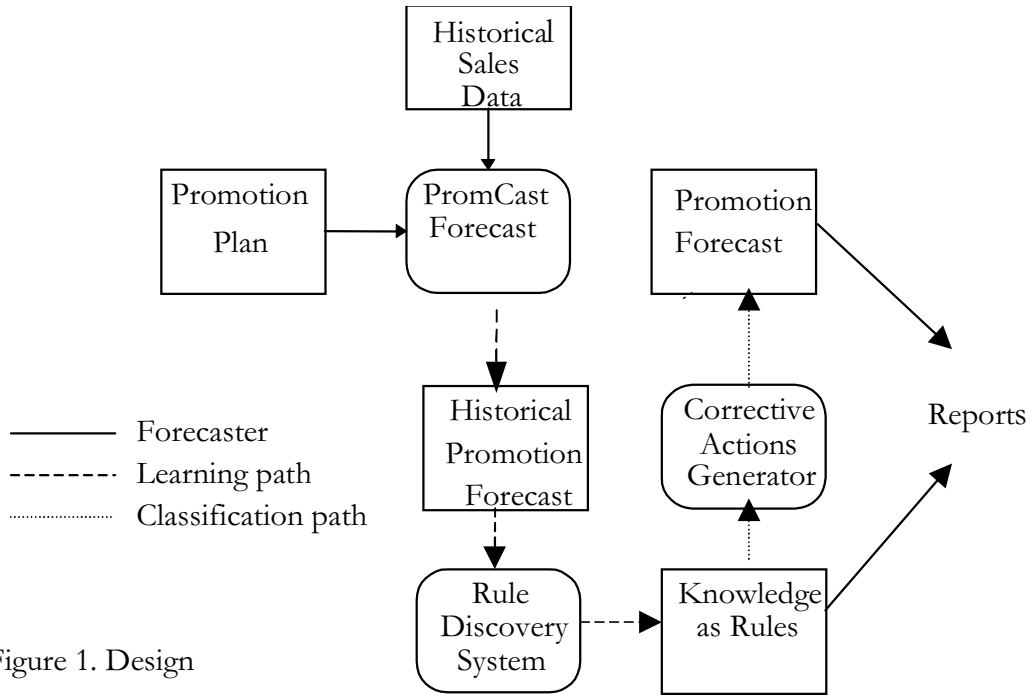

Figure 1. Design

Figure 2

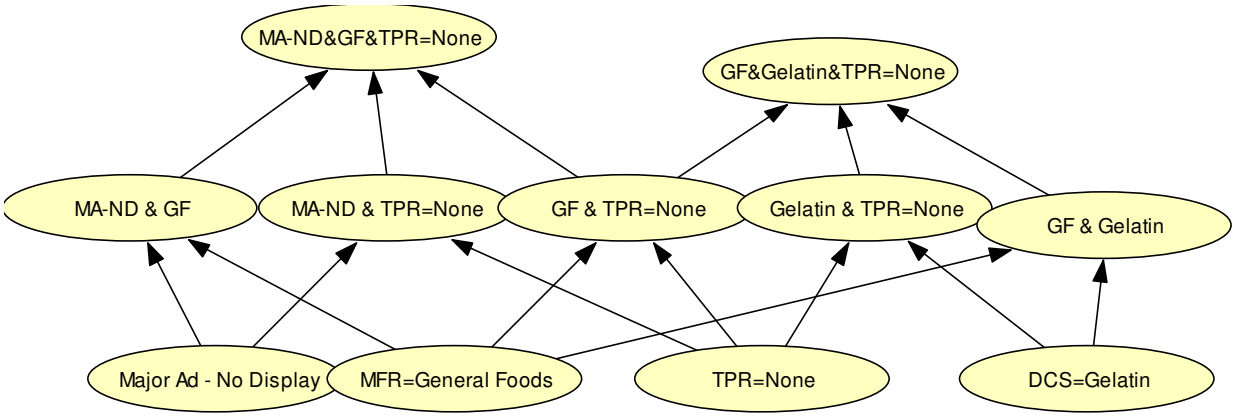
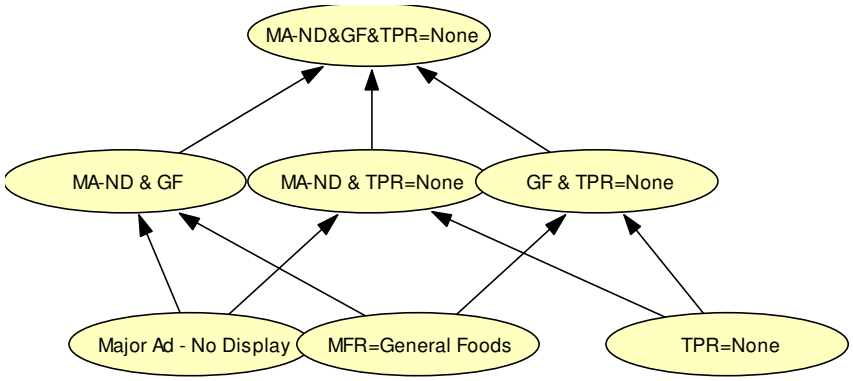
Figure 3