Bayesian Classifiers Programmed in SQL

Carlos Ordonez, Sasi K. Pitchaimalai
University of Houston
Houston, TX 77204, USA

***Abstract*—The Bayesian classifier is a fundamental classification technique. In this work, we focus on programming Bayesian classifiers in SQL. We introduce two classifiers: Naive Bayes and a classifier based on class decomposition using K-means clustering. We consider two complementary tasks: model computation and scoring a data set. We study several layouts for tables and several indexing alternatives. We analyze how to transform equations into efficient SQL queries and we introduce several query optimizations. We conduct experiments with real and synthetic data sets to evaluate classification accuracy, query optimizations and scalability. Our Bayesian classifier is more accurate than Naive Bayes and decision trees. Distance computation is significantly accelerated with horizontal layout for tables, denormalization and pivoting. We also compare Naive Bayes implementations in SQL and C++: SQL is about four times slower. Our Bayesian classifier in SQL achieves high classification accuracy, can efficiently analyze large data sets and has linear scalability.**

I. Introduction

Classification is a fundamental problem in machine learning and statistics [2]. Bayesian classifiers stand out for their robustness, interpretability, and accuracy [2]. They are deeply related to maximum likelihood estimation and discriminant analysis [2], highlighting their theoretical importance. In this work we focus on Bayesian classifiers considering two variants: Naïve Bayes [2] and a Bayesian classifier based on class decomposition using clustering [7].

In this work we integrate Bayesian classification algorithms into a DBMS. Such integration allow users to directly analyze data sets inside the DBMS and to exploit its extensive capabilities (storage management, querying, concurrency control, fault tolerance, security). We use SQL queries as the programming mechanism, since it is the standard language in a DBMS. More importantly, using SQL eliminates the need to understand and modify the internal source, which is a difficult task. Unfortunately, SQL has two important drawbacks: it has limitations to manipulate vectors and matrices and it has more overhead than a systems language like C. Keeping those issues in mind, we study how to evaluate mathematical equations with several tables layouts and optimized SQL queries.

Our contributions are the following. We present two efficient SQL implementations of Naïve Bayes for numeric and discrete attributes. We introduce a classification algorithm that builds one clustering model per class, which is a generalization of K-means [1], [4]. Our main contribution is a Bayesian classifier programmed in SQL, extending Naïve Bayes, which uses K-means to decompose each class into clusters. We generalize queries for clustering adding a new problem dimension. That is, our novel queries combine three dimensions: attribute, cluster and class subscripts. We identify Euclidean distance as the most time-consuming computation. Thus we introduce several schemes to efficiently compute distance considering different storage layouts for the data set and classification model. We also develop query optimizations that may applicable to other distance-based algorithms. A horizontal layout of the cluster centroids table and a denormalized table for sufficient statistics are essential to accelerate queries. Past research has shown the main alternatives to integrate data mining algorithms without modifying DBMS source code are SQL queries and User-Defined Functions (UDFs), being UDFs generally more efficient [5]. We show SQL queries are more efficient than UDFs for Bayesian classification. Finally, despite the fact that C++ is a significantly faster alternative, we provide evidence exporting data sets is a performance bottleneck.

The article is organized as follows. Section II introduces definitions. Section III introduces two alternative Bayesian classifiers in SQL. Section IV presents an experimental evaluation on accuracy, optimizations and scalability. Related work is discussed in Section V. Section VI concludes the paper.

http://doi.ieeecomputersociety.org/10.1109/TKDE.2009.127

II. Definitions

We focus on computing classification models on a data set $X = \{x_1, \ldots, x_n\}$ with d attributes $X_1, \ldots, X_d$, one discrete attribute G (class or target) and n records (points). We assume G has $m = 2$ values. Data set X represents a $d \times n$ matrix, where x_i represents a column vector. We study two complementary models: (1) each class is approximated by a normal distribution or histograms; (2) fitting a mixture model with k clusters on each class with K-means. We use subscripts i, j, h, g as follows: $i = 1 \ldots n, j = 1 \ldots k, h = 1 \ldots d, g = 1 \ldots m$. The T superscript indicates matrix transposition.

Throughout the article we will use a small running example, where $d = 4$, $k = 3$ (for K-means) and $m = 2$ (binary).

III. Bayesian Classifiers Programmed in SQL

We introduce two classifiers: Naïve Bayes (NB) and a Bayesian Classifier based on K-means (BKM) that performs class decomposition. We consider two phases for both classifiers: (1) computing the model; (2) scoring a data set.

A. Naïve Bayes

We consider two versions of NB: one for numeric attributes and another for discrete attributes. Numeric NB will be improved with class decomposition.

NB assumes attributes are independent and thus the joint class conditional probability can be estimated as the product of probabilities of each attribute [2]. We now discuss NB based on a multivariate Gaussian. NB has no input parameters. Each class is modeled as a single normal distribution with mean vector C_g and a diagonal variance matrix R_g. Scoring assumes a model is available and there exists a data set with the same attributes in order to predict class G. Variance computation based on sufficient statistics [5] in one pass can be numerically unstable when the variance is much smaller compared to large attribute values or when the data set has a mix of very large and very small numbers. For ill-conditioned data sets the computed variance can be significantly different from the actual variance or even become negative. Therefore, the model is computed in two passes: a first pass to get the mean per class and a second one to compute the variance per class. The mean per class is given by $C_g = \sum_{x_i \in Y_g} x_i/N_g$, where $Y_g \subseteq X$ are the records in class g. Equation $R_g = 1/N_g \sum_{x_i \in Y_g}^{n} (x_i - C_g)(x_i - C_g)^T$ gives a diagonal variance matrix R_g, which is numerically stable, but requires two passes over the data set.

The SQL implementation for numeric NB follows the mean and variance equations introduced above. We compute three aggregations grouping by g with two queries. The first query computes the mean C_g of class g with a sum(X_h)/count(*) aggregation and class priors π_g with a count() aggregation. The second query computes R_g with sum($(X_h - \mu_h)^2$). Notice the joint probability computation is not done in this phase. Scoring uses the Gaussian parameters as input to classify an input point to the most probable class, with one query in one pass over X. Each class probability is evaluated as a Gaussian. To avoid numerical issues when a variance is zero the probability is set to 1 and the joint probability is computed with a sum of probability logarithms instead of a product of probabilities. A CASE statement pivots probablities and avoid a max() aggregation. A final query determines the predicted class, being the one with maximum probability, obtained with a CASE statement.

We now discuss NB for discrete attributes. For numeric NB we used Gaussians because they work well for large data sets and because they are easy to manipulate mathematically. That is, NB does not assume any specific probability density function (pdf). Assume $X_1, \ldots, X_d$ can be discrete or numeric. If an attribute X_h is discrete (categorical) NB simply computes its histogram: probabilities are derived with counts per value divided by the corresponding number of points in each class. Otherwise, if the attribute X_h is numeric then binning is required. Binning requires two passes over the data set, pretty much like numeric NB. In the first pass bin boundaries are determined, On the second pass one-dimensional frequency histograms are computed on each attribute. The bin boundaries (interval ranges) may impact the accuracy of NB due to skewed distributions or extreme values. Thus we consider two techniques to bin attributes: (1) creating k uniform intervals between min and max; (2) taking intervals around the mean based on multiples of the standard deviation, getting more evenly populated bins. We do not study other binning schemes such as quantiles (i.e. equidepth binning).

The implementation in SQL of discrete NB is straightforward. For discrete attributes no preprocessing is required. For numeric attributes the minimum, maximum and mean can be determined in one pass in a single query. The variance for all numeric attributes is computed on a second pass to avoid numerical issues. Then each attribute is discretized finding the interval for each value. Once we have a binned version of X then we compute histograms on each attribute with SQL aggregations. Probabilities are obtained dividing by the number of records in each class. Scoring requires determining the interval for each attribute value and retrieving its probability. Each class probability is also computed by adding logarithms. NB has an advantage over other classifiers: it can handle a data set with mixed attribute types (i.e. discrete and numerical).

B. Bayesian Classifier based on K-means

We now present BKM, a Bayesian classifier based on class decomposition obtained with the K-means algorithm. BKM is a generalization of NB, where NB has one cluster per class and the Bayesian classifier has $k > 1$ clusters per class. For ease of exposition and elegance we use the same k for each class, but we experimentally show it is a sound decision. We consider two major tasks: model computation and scoring a data set based on a computed model. The most time-consuming phase is building the model. Scoring is equivalent to an E step from K-means executed on each class.

Mixture Model and Decomposition Algorithm

We fit a mixture of k clusters to each class with K-means [4]. The output are priors π (same as NB), mk weights (W), mk centroids (C) and mk variance matrices (R). We generalize NB notation with k clusters per class g with notation $g : j$, meaning the given vector or matrix refers to jth cluster from class g. Sums are defined over vectors (instead of variables). Class priors are analogous to NB: $\pi_g = N_g/n$.

We now introduce sufficient statistics in the generalized notation. Let $X_{g:j}$ represent partition j of points in class g (as determined by K-means). Then L, the linear sum of points is: $L_{g:j} = \sum_{x_i \in X_{g:j}} x_i$, and the sum of "squared" points for cluster $g : j$ becomes: $Q_{g:j} = \sum_{x_i \in X_{g:j}} x_i x_i^T$. Based on L, Q the Gaussian parameters per class g are:

$$C_{g:j} = \frac{1}{N_{g:j}} L_{g:j}, \tag{1}$$

$$R_{g:j} = \frac{1}{N_{g:j}} Q_{g:j} - \frac{1}{(N_{g:j})^2} L_{g:j} (L_{g:j})^T. \tag{2}$$

We generalize K-means to compute m models, fitting a mixture model to each class. K-means is initialized and then it iterates until it converges on all classes. The algorithm is:

Initialization:

1) Get global N, L, Q and μ, σ.
2) Get k random points per class to initialize C.

While not all m models converge:

TABLE I
BKM TABLES.

Table	Content	PK	non-key columns
XH	Normalized data	i	$g, X_1, \ldots, X_d$
CH	Centroids	g	$C_{11}, C_{12}, \ldots, C_{dk}$
XD	distances	i, g	$d_1, \ldots, d_k$
XN	nearest cluster	i, g	j
NLQ	Suff. stats.	g, j	$N_g, L_1, \ldots, L_d, Q_1, \ldots, Q_d$
WCR	mixture model	g, j	prior,$C_1, \ldots, C_d, R_1, \ldots, R_d$
XP	prob. per cluster	i, g	$p_1, \ldots, p_k$
XC	predicted class	i	g
MQ	model quality, N_g	-	

1) E step:
   get k distances j per g; find nearest cluster j per g;
   update N, L, Q per class.
2) M step:
   update W, C, R from N, L, Q per class;
   compute model quality per g; monitor convergence.

For scoring, in a similar manner to NB, a point is assigned to the class with highest probability. In this case this means getting the most probable cluster based on mk Gaussians. The probability gets multiplied by the class prior by default (under the common Bayesian framework), but it can be ignored for imbalanced classification problems.

Bayesian Classifier programmed in SQL

Based on the mathematical framework introduced in the previous section we now study how to create an efficient implementation of BKM in SQL.

Table I provides a summary of all working tables used by BKM. In this case the tables for NB are extended with the j subscript and since there is a clustering algorithm behind we introduce tables for distance computation. Like NB, since computations require accessing all attributes/distances/probabilities for each point at one time tables have a physical organization that stores all values for point i on the same address and there is a primary index on i for efficient hash join computation. In this manner, we force the query optimizer to perform n I/Os instead of the actual number of rows. This storage and indexing optimization is essential for all large tables having n or or more rows.

This is a summary of optimizations. There is a single set of tables for the classifier: all m models are updated on the same table scan; this eliminates the need to create multiple temporary tables. We introduce a fast distance computation mechanism based on a flattened version of the centroids; temporary tables have less rows. We use a CASE statement to get closest cluster avoiding the use of standard SQL aggregations; a join between two large tables is avoided. We delete points from classes whose model has converged; the algorithm works with smaller tables as models converge.

Model Computation

Table CH is initialized with k random points per class; this initialization requires building a stratified sample per class.

The global variance is used to normalize X to follow a $N(0, 1)$ pdf. The global mean and global variance are derived as in NB. In this manner K-means computes Euclidean distance with all attributes being on the same relative scale. The normalized data set is stored in table XH, which also has n rows. BKM uses XH thereafter at each iteration.

TABLE II
QUERY OPTIMIZATION: DISTANCE COMPUTATION.

Distance scheme	I/Os
Completely horizontal	n
Horizontal (default)	$2n$
Horizontal temp	$2kn + 2n$
Horizontal nested query	$2kn + 2n$
Hybrid vertical	dn
Vertical	dkn

We first discuss several approaches to compute distance for each class. Based on these approaches we derive a highly efficient scheme to compute distance. The following statements assume the same k per class.

The simplest way, but inefficient, way to compute distances is to create pivoted versions of X and C having one value per row. Such table structure enables the usage of SQL standard aggregations (e.g. sum(), count()). Evaluating a distance query involves creating a temporary table with $mdkn$ rows, which will be much larger than X. Since the vertical variant is expected to be the slowest it is not analyzed in the experimental section. Instead, we use a hybrid distance computation in which we have k distances per row. We start by considering a pivoted version of X called XV. The corresponding SQL query computes k distances per class using an intermediate table having dn rows. We call this variant hybrid vertical.

We consider a horizontal scheme to compute distances. All k distances per class are computed as SELECT terms. This produces a temporary table with mkn rows. Then the temporary table is pivoted and aggregated to produce a table having mn rows but k columns. Such layout enables efficient computation of the nearest cluster in a single table scan. The query optimizer decides which join algorithm to use and does not index any intermediate table. We call this approach nested horizontal. We introduce a modification to the nested horizontal approach in which we create a primary index on the temporary distance table in order to enable fast join computation. We call this scheme horizontal temporary.

We introduce a yet more efficient scheme to get distance using a flattened version of C which we call CH, that has dk columns corresponding to all k centroids for one class. The SQL code pairs each attribute from X with the corresponding attribute from the cluster. Computation is efficient because CH has only m rows and the join computes a table with n rows (for building the model) or mn rows for scoring. We call this approach the horizontal approach. There exists yet another "completely horizontal" approach in which all mk distances are stored on the same row for each point and C has only one row but mdk columns (or m tables, with dk columns each). Table II shows expected I/O cost.

The following SQL statement computes k distances for each point, corresponding to the gth model. This statement is also used for scoring and therefore it is convenient to include g.

```
INSERT INTO XD
```

```
SELECT i,XH.g  ,(X1-C1_X1)**2 + .. +(X4-C1_X4)**2,
             ..,(X1-C3_X1)**2 + .. +(X4-C3_X4)**2
FROM XH,CH WHERE XH.g=CH.g;
```

We also exploited UDFs which represent an extensibility mechanism [8] provided by the DBMS. UDFs are programmed in the fast C language and they can take advantage of arrays and flow control statements (loops, if-then). Based on this layout we developed a UDF which receives C_g as parameter and can internally manipulate it as a matrix. The UDF is called mn times and it returns the closest cluster per class. That is, the UDF computes distances and determines the closest cluster. In the experimental section we will study which is the most efficient alternative between SQL and the UDF. A big drawback about UDFs is that they are dependent on the database system architecture. Therefore, they are not portable and optimizations developed for one database system may not work well in another one.

At this point we have computed k distances per class and we need to determine the closest cluster. There are two basic alternatives: pivoting distances and using SQL standard aggregations, using case statements to determine the minimum distance. For the first alternative XD must be pivoted into a bigger table. Then the minimum distance is determined using the min() agregation. The closest cluster is the subscript of the minimum distance, which is determined joining XH. In the second alternative we just need to compare every distance against the rest using a CASE statement. Since the second alternative does not use joins and is based on a single table scan it is much faster than using a pivoted version of XD. The SQL to determine closest cluster per class for our running example is below. This statement can also used for scoring.

```
INSERT INTO XN SELECT i,g
      ,CASE WHEN d1<=d2 AND d1<=d3 THEN 1
            WHEN d2<=d3 THEN 2
            ELSE 3 END AS j
FROM XD;
```

Once we have determined the closest cluster for each point on each class, we need to update sufficient statistics, which are just sums. This computation requires joining XH and XN and partitioning points by class and cluster number. Since table NLQ is denormalized the jth vector and the jth matrix are computed together. The join computation is demanding since we are joining two tables with $O(n)$ rows. For BKM it is unlikely variance can have numerical issues, but feasible. In such case sufficient statistics are substituted by a two pass computation like NB.

```
INSERT INTO NLQ SELECT XH.g,j
  ,sum(1.0)   as Ng
  ,sum(X1)    , .. ,sum(X4)       /* L */
  ,sum(X1**2) , .. ,sum(X4**2) /* Q*/
FROM XH,XN WHERE XH.i=XN.i GROUP BY XH.g,j;
```

We now discuss the M step to update W, C, R. Computing WCR from NLQ is straightforward since both tables have the same structure and they are small. There is a Cartesian product between NLQ and the model quality table, but this latter table has only one row. Finally, to speedup computations we delete points from XH for classes whose model has converged: a reduction in size is propagated to the nearest cluster and sufficient statistics queries.

```
INSERT INTO WCR SELECT
    NLQ.g,NLQ.j
   ,Ng/MODEL.n                                  /* pi */
   ,L1/Ng, ..,L4/Ng                             /* C */
   ,Q1/Ng-(L1/Ng)**2,..,Q4/Ng-(L4/Ng)**2  /* R */
FROM NLQ,MODEL WHERE NLQ.g=MODEL.g;
```

Scoring

We consider two alternatives: based on probability (default) or distance. Scoring is similar to the E step, but there are differences. First, the new data set must be normalized with the original variance used to build the model. Such variance should be similar to the variance of the new data set. Second, we need to compute mk probabilities (distances), instead of only k distances because we are trying to find the most probable (closest) cluster among all (this is a subtle, but important difference in SQL code generation); The join condition between XH and CH gets eliminated. Third, once we have mk probabilities (distances) we need to pick the maximum (minimum) one and then the point is assigned to the corresponding cluster. To have a more elegant implementation we predict the class in two stages: we first determine the most probable cluster per class and then we compare the probabilities of such clusters. Column XH.g is not used for scoring purposes, but it is used to measure accuracy, when known.

IV. Experimental Evaluation

We analyze three majors aspects: (1) classification accuracy, (2) query optimization, (3) time complexity and speed. We compare accuracy of NB, BKM and decision trees (DT).

A. Setup

We used the Teradata DBMS running on a server with a 3.2 GHz CPU, 2 GB of RAM and a 750 GB disk.

Parameters were set as follows. We set $\epsilon = 0.001$ for K-means. The number of clusters per class was $k = 4$ (setting experimentally justified). All query optimizations were turned on by default (they do not affect model accuracy). Experiments with DTs were performed using a data mining tool.

We used real data sets to test classification accuracy (from the UCI repository) and synthetic data sets to analyze speed (varying d, n). Real data sets include pima ($d = 6, n = 768$), spam ($d = 7, n = 4601$), bscale ($d = 4, n = 625$), wbcancer ($d = 7, n = 569$). Categorical attributes (≥ 3 values) were transformed into binary attributes.

B. Model Accuracy

In this section we measure the accuracy of predictions when using Bayesian classification models. We used 5-fold cross-validation. For each run the data set was partitioned into a training set and a test set. The training set was used to compute the model, whereas the test set was used to independently measure accuracy. The training set size was 80% and the test set was 20%. BKM ran until K-means converged on all classes. Decision tress (DTs) used the CN5.0 algorithm splitting nodes until they reached a minimum percentage of purity or became

TABLE III
ACCURACY VARYING k.

Dataset	$k=2$	$k=4$	$k=6$	$k=8$	$k=16$
pima	72%	74%	72%	71%	75%
spam	75%	75%	64%	72%	52%
bscale	55%	59%	56%	60%	65%
wbcancer	93%	92%	93%	92%	89%

TABLE IV
COMPARING ACCURACY: NB, BKM AND DT.

Dataset	Algorithm	Global	Class-0	Class-1
pima	NB	76%	80%	68%
	BKM	76%	87%	53%
	DT	68%	76%	53%
spam	NB	70%	87%	45%
	BKM	73%	91%	43%
	DT	80%	85%	72%
bscale	NB	50%	51%	30%
	BKM	59%	59%	60%
	DT	89%	96%	0%
wbcancer	NB	93%	91%	95%
	BKM	93%	84%	97%
	DT	95%	94%	96%

too small. Pruning was applied to reduce overfit. The number of clusters for BKM by default was $k=4$.

The number of clusters (k) is the most important parameter to tune BKM accuracy. Table III shows accuracy behavior as k increases. A higher k produces more complex models. Intuitively, a higher k should achieve higher accuracy because each class can be better approximated with localized clusters. On the other hand, a higher k than necessary can lead to empty clusters. As can be seen a lower k produces better models for spam and wbcancer, whereas a higher k produces higher accuracy for pima and bscale. Therefore, there is no ideal setting for k, but in general $k \leq 20$ produces good results. Based on these results we set $k=4$ by default since it provides reasonable accuracy for all data sets.

Table IV compares the accuracy of the three models, including the overall accuracy as well as a breakdown per class. Accuracy per predicted class is important to understand issues with imbalanced classes and detecting subsets of highly similar points. In practice one class is generally more important than the other one (asymmetric), depending if the classification task is to minimize false positives or false negatives. This is a summary of findings. First of all, considering global accuracy BKM is better than NB on two data sets and becomes tied with the other two data sets. On the other hand, compared to decision trees it is better in one case and worse otherwise. NB follows the same pattern, but showing a wider gap. However, it is important to understand why BKM and NB perform worse than DT in some cases, looking at the accuracy breakdown. In this case, BKM generally does better than NB, except in two cases. Therefore, class decomposition does increase prediction accuracy. On the other hand, compared to decision trees, in all cases where BKM is less accurate than DT on global accuracy BKM has higher accuracy on one class. In fact, on closer look it can be seen that DT completely misses prediction for one class for the bscale data set. That is, global accuracy for DT is misleading. We conclude that BKM performs better than NB and it is competitive with DT.

TABLE V
QUERY OPTIMIZATION: DISTANCE COMPUTATION $n = 100$K (SECS).

	$d=8$		$d=16$	
Distance scheme	$k=8$	$k=16$	$k=8$	$k=16$
Horizontal	2	7	2	7
Horizontal temp	80	211	99	225
Horizontal nested q.	201	449	311	544
Hybrid Vertical	84	155	146	155

TABLE VI
QUERY OPTIMIZATION: SQL VERSUS UDF (SCORING, $n = 1$M).

	$d=4$		$d=8$	
k	SQL	UDF	SQL	UDF
2	23	36	34	61
4	31	47	42	71
6	40	55	61	85
8	56	62	66	111

C. Query Optimization

Table V provides detailed experiments on distance computation, the most demanding computation for BKM. Our best distance strategy is two orders of magnitude faster than the worst strategy and it is one order of magnitude faster than its closest rival. The explanation is that I/O is minimized to n operations and computations happen in main memory for each row through SQL arithmetic expressions. Notice a standard aggregation on the pivoted version of X (XV) is faster than the horizontal nested query variant.

Table VI compares SQL with UDFs to score the data set (computing distance and finding nearest cluster per class). We exploit scalar UDFs [5]. Since finding the nearest cluster is straightforward in the UDF this comparison considers both computations as one. This experiment favors the UDF since SQL requires accessing large tables in separate queries. As we can see SQL (with arithmetic expressions) turned out be faster than the UDF. This was interesting because both approaches used the same table as input and performed the same I/O reading a large table. We expected SQL to be slower because it required a join and XH and XD were accessed. The explanation was that the UDF has overhead to pass each point and model as parameters in each call.

D. Speed and Time Complexity

We compare SQL and C++ running on the same computer. We also compare the time to export with ODBC. C++ worked on flat files exported from the DBMS. We used binary files in order to get maximum performance in C++. Also, we shut down the DBMS when C++ was running. In short, we conducted a fair comparison. Table VII compares SQL, C++ and ODBC varying n. Clearly, ODBC is a bottleneck. Overall,

TABLE VII
COMPARING SQL, C++ AND ODBC WITH NB ($d=4$). TIMES IN SECS.

$n \times 1M$	SQL	C++	ODBC
1	10	2	510
2	19	5	1021
4	37	9	2041
8	76	18	4074
16	119	36	*

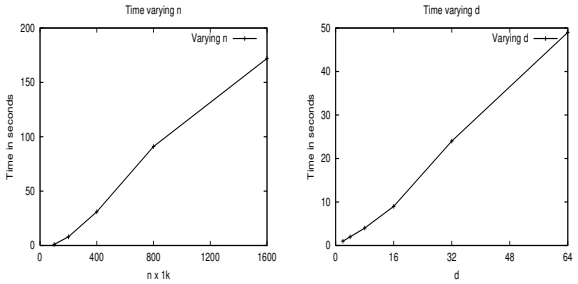

Fig. 1. BKM Classifier: Time complexity; ($d = 4, k = 4, n = 100$k).

both languages scale linearly. We can see SQL performs better as n grows because DBMS overhead becomes less important. However, C++ is about four times faster.

Figure 1 shows BKM time complexity varying n, d with large data sets. Time is measured for one iteration. BKM is linear in n and d, highlighting its scalability.

V. Related Work

The most widely used approach to integrate data mining algorithms into a DBMS is to modify the internal source code. Scalable K-means (SKM) [1] and O-cluster [3] are two examples of clustering algorithms internally integrated with a DBMS. A discrete Naive Bayes classifier, has been internally integrated with the SQL Server DBMS. On the other hand, the two main mechanisms to integrate data mining algorithms without modifying the DBMS source code are SQL queries and UDFs. A discrete Naïve Bayes classifier programmed in SQL is introduced in [6]. We summarize differences with ours. The data set is assumed to have discrete attributes: binning numeric attributes is not considered. It uses an inefficient large pivoted intermediate table, whereas our discrete NB model can directly work on a horizontal layout. Our proposal extends clustering algorithms in SQL [4] to perform classification, generalizing Naïve Bayes [2]. K-means clustering was programmed with SQL queries introducing three variants [4]: standard, optimized and incremental. We generalized the optimized variant. Notice classification represents a significantly harder problem than clustering. User-Defined Functions (UDFs) are identified as an important extensibility mechanism to integrate data mining algorithms [5], [8]. ATLaS [8] extends SQL syntax with object-oriented constructs to define aggregate and table functions (with initialize, iterate and terminate clauses), providing a user-friendly interface to the SQL standard. We point out several differences with our work. First, we propose to generate SQL code from a host language, thus achieving Turing-completeness in SQL (similar to embedded SQL). We showed the Bayesian classifier can be solved more efficiently with SQL queries than with UDFs. Even further, SQL code provides better portability and ATLaS requires modifying the DBMS source code. Class decomposition with clustering is shown to improve NB accuracy [7]. The classifier can adapt to skewed distributions and overlapping subsets of points by building better local models. In [7] EM was the algorithm to fit a mixture per class. Instead, we decided to use K-means because it is faster, simpler, better understood in database research and has less numeric issues.

VI. Conclusions

We presented two Bayesian classifiers programmed in SQL: the Naïve Bayes (NB) classifier (with discrete and numeric versions) and a generalization of Naïve Bayes (BKM), based on decomposing classes with K-means clustering. We studied two complementary aspects: increasing accuracy and generating efficient SQL code. We introduced query optimizations to generate fast SQL code. The best physical storage layout and primary index for large tables is based on the point subscript. Sufficient statistics are stored on denormalized tables. Euclidean distance computation uses a flattened (horizontal) version of the cluster centroids matrix, which enables arithmetic expressions. The nearest cluster per class, required by K-means, is efficiently determined avoiding joins and aggregations. Experiments with real data sets compared NB, BKM and decision trees. The numeric and discrete versions of NB had similar accuracy. BKM was more accurate than NB and was similar to decision trees in global accuracy. However, BKM was more accurate when computing a breakdown of accuracy per class. A low number of clusters produced good results in most cases. We compared equivalent implementations of NB in SQL and C++ with large data sets: SQL was four times slower. SQL queries were faster than UDFs to score, highlighting the importance of our optimizations. NB and BKM exhibited linear scalability in data set size and dimensionality.

There are many opportunities for future work. We want to derive incremental versions or sample-based methods to accelerate the Bayesian classifier. We want to improve our Bayesian classifier to produce more accurate models with skewed distributions, data sets with missing information and subsets of points having significant overlap with each other, which are known issues for clustering algorithms. We are interested in combining dimensionality reduction techniques like PCA or factor analysis with Bayesian classifiers. UDFs need further study to accelerate computations and evaluate complex mathematical equations.

References

[1] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. ACM KDD Conference*, pages 9–15, 1998.

[2] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.

[3] B.L. Milenova and M.M. Campos. O-cluster: Scalable clustering of large high dimensional data sets. In *Proc. IEEE ICDM conference*, page 290, Washington, DC, USA, 2002. IEEE Computer Society.

[4] C. Ordonez. Integrating K-means clustering with a relational DBMS using SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 18(2):188–201, 2006.

[5] C. Ordonez. Building statistical models and scoring with UDFs. In *Proc. ACM SIGMOD Conference*, pages 1005–1016, 2007.

[6] S. Thomas and M.M. Campos. SQL-based Naive Bayes model building and scoring. *US Patent and Trade Office*, US Patent 7051037, 2006.

[7] R. Vilalta and I. Rish. A decomposition of classes via clustering to explain and improve Naive Bayes. In *Proc. ECML Conference*, pages 444–455, 2003.

[8] H. Wang, C. Zaniolo, and C.R. Luo. ATLaS: A small but complete SQL extension for data mining and data streams. In *Proc. VLDB Conference*, pages 1113–1116, 2003.