Detailed In-database Prediction Example

This section provides a detailed example of doing in-database prediction on a publicly available data set using `hpdglm()`. This example uses marketing data from a banking institution. The original study using this data is available at https://archive.ics.uci.edu/ml/datasets/Bank+Marketing. The goal of this exercise is to determine if a phone-based marketing campaign will be successful based on several data points based on customer metrics and details about the call (time of day, etc.). This is a classification problem and the dependent variable is the variable "y". A "yes" indicates that the call was successful in getting the customer to open a bank account. A "no" indicates that the marketing attempt was not successful.

Outline of the analytical process:

1. Obtain the data set.
2. Create a table in HP Vertica to hold the original data.
3. Explore and Prepare the data in HP Vertica and HP Distributed R.
4. Separate the data into Training and Testing sets.
5. Create the model in HP Distributed R and deploy to HP Vertica.
6. Apply the model to data in your HP Vertica tables.
7. Check model accuracy.

Obtaining the Data Set:

1. First, obtain the original data set from https://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank-additional.zip and copy it to a HP Vertica database node:

   You can use `wget` to copy the data to your node:

   ```
   wget https://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank-
   additional.zip
   ```

2. Unzip the file:

   ```
   unzip bank-additional.zip
   ```

   The zip is extracted into a new folder named *bank-additional*. There are two files in the directory; *bank-additional.csv* and *bank-additional-full.csv*. This exercise uses ***bank-additional-full****.csv* as it contains the full data set.

Creating a table in HP Vertica to hold the original data and load the data:

1. Create a table on HP Vertica to hold the original data set:

```
CREATE TABLE IF NOT EXISTS bank_original (
    age int, job varchar, marital varchar,
    education varchar, "default" varchar, housing varchar,
    loan varchar, contact varchar, MONTH varchar,
    day_of_week char(5), duration int, campaign int,
    pdays int, "previous" int, poutcome varchar,
    "emp.var.rate" float, "cons.price.idx" float, "cons.conf.idx" float,
    euribor3m float, "nr.employed" float, y varchar(5));
```

2. Load the data in HP Vertica

```
COPY bank_original FROM '/home/dbadmin/bank-additional/bank-additional-full.csv'
DELIMITER ';';
```

   41188 rows should be loaded. If only 4119 rows are loaded then you loaded the wrong data set. Be sure you load the full data set.

Exploring and preparing the data in HP Vertica and HP Distributed R:

Before creating the model, you need to complete a few steps to explore and prepare the data.

1. First, create a normalized view using the z-score (standard score) of the numerical columns. You normalize the numerical values to a standard score to give all of the columns equal weighting when you run logistic regression. This prevents the *nr.employed* value, which is in the thousands, from "drowning out" the *age* value, whose median score is 38. The z-score is obtained by subtracting the column mean from the row value and dividing by the standard deviation of the column. For example, if the age was 45, then you could determine the z-score for 45 years with the command: `select (45 - avg(age))/stddev(age) from bank_original;`.

   The following query calculates the z-scores for all numerical values in the data set and stores them in a view named *bank_normalized*. The z-index columns are renamed with _z at the end of the column name. At the same time, replace the period with underscore in the column names to avoid any confusion in the next steps.

Note: This step also discards the *duration* feature as advised on the web pagefor the data set.

```
CREATE VIEW bank_normalized AS
    SELECT (age-m_age)/std_age AS age_z,
       job,
       marital,
       education,
       "default",
       housing,
       loan,
       contact,
       MONTH,
       day_of_week,
       (campaign-m_campaign)/std_campaign AS campaign_z,
       (pdays-m_pdays)/std_pdays AS pdays_z,
       (previous-m_previous)/std_previous AS previous_z,
       poutcome,
       ("emp.var.rate"-m_emp_var_rate)/std_emp_var_rate AS emp_var_rate_z,
       ("cons.price.idx"-m_cons_price_idx)/std_cons_price_idx AS cons_price_idx_z,
       ("cons.conf.idx"-m_cons_conf_idx)/std_cons_conf_idx AS cons_conf_idx_z,
       (euribor3m-m_euribor3m)/std_euribor3m AS euribor3m_z,
       ("nr.employed"-m_nr_employed)/std_nr_employed AS nr_employed_z,
       y
    FROM bank_original ,
    (SELECT avg(age) m_age,
        stddev(age) std_age,
        avg(campaign) m_campaign,
        stddev(campaign) std_campaign,
        avg(pdays) m_pdays,
        stddev(pdays) std_pdays,
        avg(previous) m_previous,
        stddev(previous) std_previous,
        avg("emp.var.rate") m_emp_var_rate,
        stddev("emp.var.rate") std_emp_var_rate,
        avg("cons.price.idx") m_cons_price_idx,
        stddev("cons.price.idx") std_cons_price_idx,
        avg("cons.conf.idx") m_cons_conf_idx,
        stddev("cons.conf.idx") std_cons_conf_idx,
        avg(euribor3m) m_euribor3m,
        stddev(euribor3m) std_euribor3m,
        avg("nr.employed") m_nr_employed,
        stddev("nr.employed") std_nr_employed
    FROM bank_original) AS tbl_mean_std;
```

2. Next, explore the categorical columns in the data. The categorical columns in this data set are:
   - job
   - marital

- education
- "default"
- housing
- loan
- contact
- month
- day_of_week
- poutcome
- y

Some of the categories in the data have a very low frequency. For example, there are only three samples with the value *yes* for the *default* feature. For huge data sets you can optimize the analysis by focusing on the top-N categories. Top-n categories are items in the category with a threshold above a certain level. The decision about the threshold for identifying low-frequency category items depends on the ratio of their frequency to the total number of samples in the data set. In the bank marketing data set, a threshold of 2000 samples (about 5% of the total samples) seems like a reasonable threshold for this exercise. For items in a category above the threshold level you keep the items labeled as-is. For items with instances below the threshold re-label them as *other*.

You can calculate the frequency of a feature with the SQL `count()` function. The examples below list the categories that have items that fall below the threshold we have set (roughly 2000 instances).

```
SELECT job, COUNT(*) AS freq FROM bank_normalized GROUP BY job ORDER BY freq DESC;

       job       |  freq
-----------------+-------
 "admin."        | 10422
 "blue-collar"   |  9254
 "technician"    |  6743
 "services"      |  3969
 "management"    |  2924
 "retired"       |  1720
 "entrepreneur"  |  1456
 "self-employed" |  1421
 "housemaid"     |  1060
 "unemployed"    |  1014
 "student"       |   875
 "unknown"       |   330
(12 rows)
```

```
SELECT marital, COUNT(*) AS freq FROM bank_normalized GROUP BY marital ORDER BY freq
DESC;

  marital   |  freq
------------+-------
 "married"  | 24928
 "single"   | 11568
 "divorced" |  4612
 "unknown"  |    80
(4 rows)

SELECT education, COUNT(*) AS freq FROM bank_normalized GROUP BY education ORDER BY
freq DESC;

       education       |  freq
-----------------------+-------
 "university.degree"   | 12168
 "high.school"         |  9515
 "bASic.9y"            |  6045
 "professional.course" |  5243
 "bASic.4y"            |  4176
 "bASic.6y"            |  2292
 "unknown"             |  1731
 "illiterate"          |    18
(8 rows)

SELECT "default", COUNT(*) AS freq FROM bank_normalized GROUP BY "default" ORDER BY
freq DESC;

  default  |  freq
-----------+-------
 "no"      | 32588
 "unknown" |  8597
 "yes"     |     3
(3 rows)

SELECT housing, COUNT(*) AS freq FROM bank_normalized GROUP BY housing ORDER BY freq
DESC;

  housing  |  freq
-----------+-------
 "yes"     | 21576
 "no"      | 18622
 "unknown" |   990
(3 rows)

SELECT loan, COUNT(*) AS freq FROM bank_normalized GROUP BY loan ORDER BY freq DESC;

   loan    |  freq
-----------+-------
 "no"      | 33950
 "yes"     |  6248
 "unknown" |   990
(3 rows)
```

```
SELECT contact, COUNT(*) AS freq FROM bank_normalized GROUP BY contact ORDER BY freq
DESC;

   contact   |  freq
-------------+-------
 "cellular"  | 26144
 "telephone" | 15044
(2 rows)

SELECT month, COUNT(*) AS freq FROM bank_normalized GROUP BY month ORDER BY freq
DESC;

 month |  freq
-------+-------
 "may" | 13769
 "jul" |  7174
 "aug" |  6178
 "jun" |  5318
 "nov" |  4101
 "apr" |  2632
 "oct" |   718
 "sep" |   570
 "mar" |   546
 "dec" |   182
(10 rows)

SELECT day_of_week, COUNT(*) AS freq FROM bank_normalized GROUP BY day_of_week ORDER
BY freq DESC;

 day_of_week | freq
-------------+------
 "thu"       | 8623
 "mon"       | 8514
 "wed"       | 8134
 "tue"       | 8090
 "fri"       | 7827
(5 rows)

SELECT poutcome, COUNT(*) AS freq FROM bank_normalized GROUP BY poutcome ORDER BY
freq DESC;

   poutcome    |  freq
---------------+-------
 "nonexistent" | 35563
 "failure"     |  4252
 "success"     |  1373
(3 rows)

SELECT y, COUNT(*) AS freq FROM bank_normalized GROUP BY y ORDER BY freq DESC;

   y   |  freq
-------+-------
 "no"  | 36548
 "yes" |  4640
(2 rows)
```

3. Once you have determined the categories for which you want to create top-N views, then you can use the decode() function to keep the category labels as-is for items that occur above the threshold and re-label category items below the threshold as *other*.

```
CREATE VIEW bank_top_n AS
SELECT age_z,
       DECODE(job, '"admin."', 'admin.', '"blue-collar"', 'blue-collar',
'"technician"', 'technician', '"services"', 'services', '"management"',
'management', 'other') AS job,
       DECODE(marital, '"married"', 'married', '"single"', 'single', 'other') AS
marital,
       DECODE(education, '"basic.6y"', 'other', '"unknown"', 'other', '"illiterate"',
'other', education) AS education,
       DECODE("default", '"no"', 'no', 'others') AS "default",
       DECODE(housing, '"yes"', 'yes', 'other') AS housing,
       DECODE(loan, '"no"', 'no', 'other') AS loan,
       contact,
       DECODE(MONTH, '"may"','may', '"jul"','jul',
'"aug"','aug','"jun"','jun','"nov"','nov', 'other') AS MONTH,
       day_of_week,
       campaign_z,
       pdays_z,
       previous_z,
       poutcome,
       emp_var_rate_z,
       cons_price_idx_z,
       cons_conf_idx_z,
       euribor3m_z,
       nr_employed_z,
       y
FROM bank_normalized;
```

4. Now you can use HP Distributed R to further prepare the data. At this point you have the original data in the table *bank_original* and two views; *bank_normalized* contains all of the data in *bank_original* but with the numerical data normalized, and *bank_top_n* contains the categorical data with low-frequency occurrences bucketed as *other* in their respective columns. The final step in preparing the data is to convert the categories in *bank_top_n* into an equivalent binary format. This is required because some algorithms, such as `hpdglm()`, do not automatically convert categorical data into binary values (also known as dummy variables). While it is possible to do this binary conversion using SQL commands, HP has created an R script named `cat2num.R` that can quickly and efficiently created a new view in HP Vertica where the categorical columns of an existing table or view are converted into multiple columns with binary values.

   For example, if you have a single column named marital, and it contains values of married and single, then to convert this into binary values you create a single new column; married_1 and

set it to a 1 if the person is married, 0 otherwise, based on the value in the marital column. You can do this for any categorical column data. For n categories you create n - 1 new columns.

Note: You must have configured your Distributed R cluster to connect to a HP Vertica database over ODBC to use the `cat2num.R` script.

To convert the categorical values into binary values:

a. Download the `cat2num.R` R script from github (https://github.com/vertica/DistributedR/blob/master/demo/example/hpdglm/cat2num.R) and copy it to your distributed R master node.

b. Open an R session on your Distributed R master and source the cat2num.R script:

```
source('/home/dbadmin/cat2num.R')
```

c. The signature of the cat2num function is:

```
cat2num (srcTable, dsn, features = list(...), except=list(...), dstTable,
view=TRUE)
```

Where the arguments of the function are:

- `srcTable`: The name of the source table or view.
- `dsn`: ODBC DSN name
- `features`: A list containing the name of columns from srcTable that are supposed to appear in the dstTable. If omitted then all columns are included.
- `except`: The list of column names that should be excluded from the dstTable.
- `dstTable`: The name of the destination table.
- `view`: When set *TRUE* (default) a view is created in the database. Otherwise a table is created.

Therefore, to call the function for the bank example, use the following command (replace *VerticaDSN* with the name of your DSN connection):

```
cat2num (srcTable='bank_top_n', dsn='VerticaDSN', dstTable='bank_top_n_num')
```

The command creates a new view in HP Vertica with the categorical columns in bank_top_n converted to binary equivalents.

Separating the data into Training and Testing sets:

Now that you have converted the categorical data into binary values you can combine the views and separate the data into training and testing sets.

1. In vsql, run the following commands to unify the different views and separate the data into training and testing sets:

```
CREATE TABLE bank_test AS
  (SELECT *
   FROM bank_top_n_num
   WHERE y_1=0
     AND RANDOM() < 0.2);

INSERT INTO bank_test
  (SELECT *
   FROM bank_top_n_num
   WHERE y_1=1
     AND RANDOM() < 0.2);

CREATE TABLE bank_training AS
  (SELECT *
   FROM bank_top_n_num EXCEPT SELECT *
   FROM bank_test);
```

The first command creates the *bank_test* table and loads a random 20% sample of the column `y_1` where it equals 0 (the independent y variable in the dataset being equal to 0). The second command inserts additional rows into the table with a random sample of `y_1` being 1 (the independent y variable in the data set being equal to 1). The third command creates a new table named *bank_training* that contains all of the data not in the *bank_test* table.

The reason you split the data this way is to get an equal distribution of values of the *y* dependent variable. By sampling 20% of each *y* feature, you put 20% of the total data into the testing set, and the remaining 80% of the data is put into the training set.

Creating the model in HP Distributed R and deploy to HP Vertica:

Now that you have the data prepped and broken into training and testing sets in HP Vertica, you can now create the model in Distributed R and deploy it to be used on HP Vertica for predictions on very large data sets. In this example, the training set is 80% of the complete data set. You could certainly do all the analysis in this particular example from within R, however when you have very large data sets it is more efficient to run the prediction function from within HP Vertica.

To create and deploy the model:

1. Log into your Distributed R master node and start an R session.
2. In your R session, run the following to load the required library, start Distributed R, and load the training data from your database. Replace the `dsn` argument with the name of your HP Vertica ODBC DSN.

```
library(HPdata)
distributedR_start()
LoadedData <-  db2darrays(tableName='bank_training', dsn='VerticaDSN', resp=list('y_
1'))
```

   After the data is loaded Distributed R responds with:

```
Loading total 31068 rows from table bank_training from Vertica with approximate
partition of 1942 rows
progress: 100%
```

3. Next create the model in Distributed R:

```
library(HPdregression)
theModel <- hpdglm(responses=LoadedData$Y, predictors=LoadedData$X, family=binomial)
```

4. Finally, use the following command to deploy the model to HP Vertica. Replace the `dsn` argument with the name of your HP Vertica ODBC DSN.

```
deploy.model(model=theModel, dsn='SF', modelName='demoModel', modelComments='A
logistic regression model for bank data')
```

Applying the model to data in your HP Vertica tables.

Now that you have deployed your model to HP Vertica you can use the HP Vertica `glmpredict()` function to apply the model to your testing data. The predict function currently requires that you specify all of the features (instead of using a select statement or '*'). The following command loads the result into a table:

```
CREATE TABLE predict_test AS
  (SELECT y_1,
          GlmPredict(age_z, job_1, job_2, job_3, job_4, job_5, marital_1, marital_2,
education_1, education_2, education_3, education_4, education_5, default_1, housing_1,
loan_1, contact_1, month_1, month_2, month_3, month_4, month_5, day_of_week_1, day_of_
```

```
week_2, day_of_week_3, day_of_week_4, campaign_z, pdays_z, previous_z, poutcome_1,
poutcome_2, emp_var_rate_z, cons_price_idx_z, cons_conf_idx_z, euribor3m_z, nr_employed_z
USING PARAMETERS model='dbadmin/demoModel' ,TYPE='response')
   FROM bank_test);
```

Checking the model accuracy:

Once you have run your prediction you can do a quick check on the accuracy by comparing the actual responses to the correct responses, and then divide by the total responses to get percent accuracy for your test set:

```
=> SELECT sum(ROUND(1-abs(y_1 - GlmPredict))) from predict_test;
 sum
------
 7249
(1 row)

=> SELECT count(*) from predict_test;
 count
-------
  8367
(1 row)

=> select 7294 / 8367 as accuracy;
       accuracy
----------------------
 0.871758097286960679
(1 row)
```

The actual numbers returned may differ slightly each time a model is created, but you can expect between 87% and 90% accuracy with this data using `hpdglm()`.