Introduction to Intel DAAL, Part 1: Polynomial Regression with Batch Mode Computation

Ryo Asai

Colfax International

October 28, 2015

Abstract

This is the part 1 of 3 of an introductory series of publications on the Intel Data Analytics Acceleration Library (DAAL). DAAL is a data analytics library optimized for modern highly parallel computer architectures such as Intel Xeon and Intel Xeon Phi processors. The goal of this series is to provide developers a technical overview for developing applications using DAAL.

In this paper we focus on two aspects of developing an application with Intel DAAL: data management and computation. As a practical example, we implement a simple machine learning application with polynomial regression using the library in the batch computation mode. We demonstrate using this application for data-based prediction of hydrodynamics properties of yachts. The source code and data for the sample application are available for free download.

The second and third part of the series will discuss other aspects of data analysis with DAAL. In part 2, we discuss distributed data and computation in conjunction with MPI. In the third part, we discuss the case with multiple data sets and interfacing with a relational database using SQL.

Table of Contents

1 Introduction **2**
2 Computation with Intel DAAL **3**
2.1 Algorithm 3
2.2 Computation Mode 3
3 Data management with Intel DAAL **4**
3.1 Data Structure 4
3.2 Loading Data 5
3.2.1 From simple arrays 5
3.2.2 From CSV files 6
3.3 Data Extraction 7
4 Example: Linear Regression **8**
4.1 How-to: Linear Regression 8
4.1.1 Batch Training 9
4.1.2 Prediction 10
4.2 Example implementation 12
4.3 Results 13
5 Closing words **14**
A Mathematics of regression **15**
A.1 Linear regression 15
A.2 Polynomial regression 16

Colfax International (http://www.colfax-intl.com/) is a leading provider of innovative and expertly engineered workstations, servers, clusters, storage, and personal supercomputing solutions. Colfax International is uniquely positioned to offer the broadest spectrum of high performance computing solutions, all of them completely customizable to meet your needs - far beyond anything you can get from any other name brand. Ready-to-go Colfax HPC solutions deliver significant price/performance advantages, and increased IT agility, that accelerates your business and research outcomes. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

1. Introduction

This is the first part of a series of publications introducing the Intel Data Analytics Acceleration Library (DAAL). Intel DAAL is a highly optimized data analytics library developed by Intel that is designed to be a complete solution for data analytics with modern highly parallel systems, such as Intel Xeon processors and Intel Xeon Phi coprocessors.

Intel DAAL covers a wide range of data analysis considerations, from data management tools to data analysis algorithms. Figure 1 is a schematic showing the building blocks available with Intel DAAL.

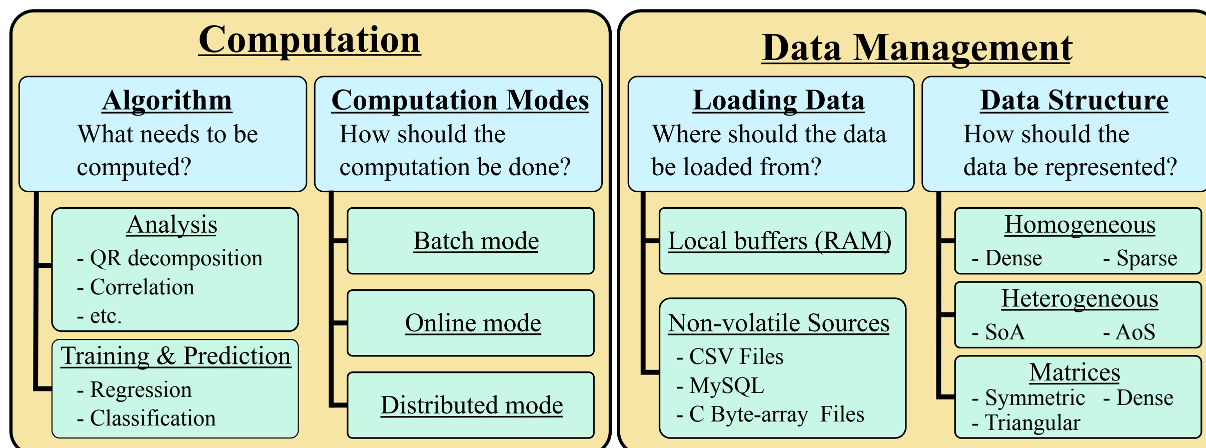

Figure 1: General overview of some of the building blocks offered by Intel DAAL.

In this series of three papers we are taking a look at three different situations. In each publication, we discuss different branches of the choices shown in Figure 1 and implement a working sample Intel DAAL application adapted to each situation.

- Part 1:
  In this paper, we discuss a very simple situation where there is a single Comma Separated Value (CSV) data file on a single compute node.
- Part 2:
  In the second paper, we will deal with a distributed computation situation where there are multiple data sets in multiple compute nodes.
- Part 3:
  In the final paper, we will be streaming data from a relational database in chunks and analyzing them as they arrive.

All three publications, and the working source code samples will be available on the Colfax Research website [1].

2. Computation with Intel DAAL

For computation with Intel DAAL, there are two major considerations to make at the start: algorithm, and the computation mode (see Figure 1).

2.1. Algorithm

Generally, the first step in planning an Intel DAAL application is to choose the algorithm. DAAL currently supports algorithms shown in Figure 2.

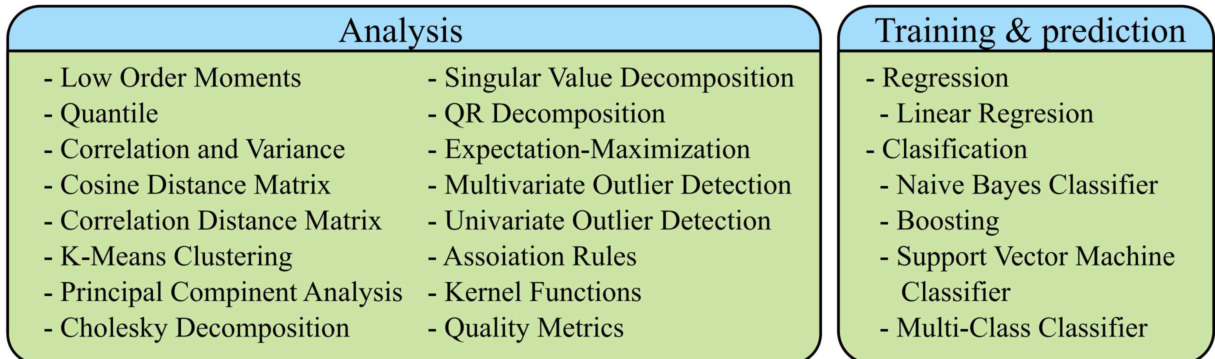

Figure 2: Algorithms currently supported by Intel DAAL.

Discussion of each individual algorithm is beyond the scope of the paper. For a brief introduction to these algorithms, see the post by James Reinders from Intel [2]. For detailed explanation of the algorithms, refer to the Intel DAAL documentation [3].

In this series, we will be using the linear regression algorithm to demonstrate the different usage models of DAAL. Linear regression is one of the basic forms of machine learning. For a brief introduction to linear regression, refer to Appendix A.

2.2. Computation Mode

Next, the developer must decide on which computation mode to use for the Intel DAAL application. The computation mode refers to how the computation is done, and the choice depends on the computing system and the environment that the application will run in.

Most algorithms in Intel DAAL support three different computation modes: batch, online and distributed. Figure 3 illustrates the usage model of three modes.

- *Batch mode:*
  Batch mode is the simplest mode which only uses a single data set. Therefore it is useful when all data exists in a single location (e.g. single file) on a single compute node.

- *Online mode:*
  Online mode supports multiple training sets. It is useful when data exists in multiple locations, or when data is only available one block at a time. Online mode is used for a single compute node.

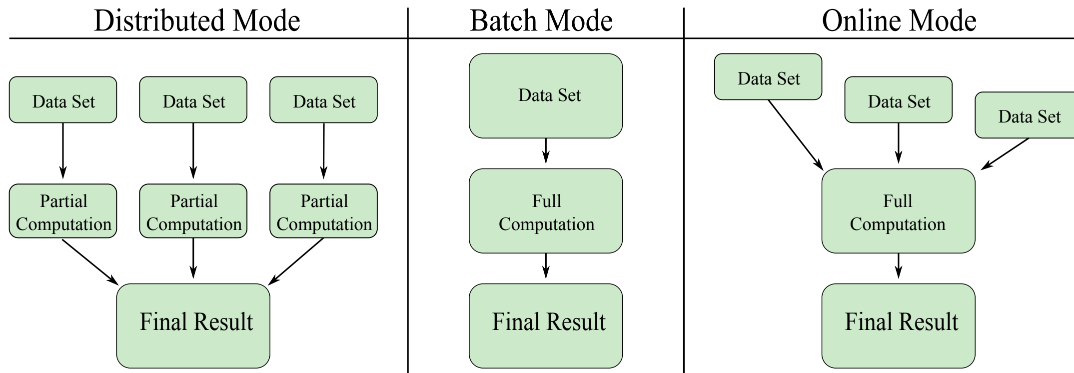

Figure 3: The three modes for computation.

- *Distributed mode:*
  Distributed mode allows computation of partial results, and supports multiple data sets. Thus this model is the recommended model when the data and computation is distributed across multiple compute nodes, for example, in an MPI or an Apache Spark environment.

Note that not all algorithms support each of these modes; consult the Intel DAAL documentation page of the algorithm to see which modes are supported by a particular algorithm. In our case, linear regression supports all three modes.

For the case studies in this series of publications, we will be applying all three modes. For the case in part 1, batch mode is best suited as there is only a single data set. For the case in part 2, the distributed data and computation calls for using the distributed mode. And finally, for the case in part 3, the online mode is most useful because the data comes in chunks.

3. Data management with Intel DAAL

In this section, we discuss some of the basics of data management. Intel DAAL supports some advanced features in data management, such as data serialization and compression. These advanced topics will be covered in upcoming publications. At the basic level, there are two considerations to make for data management: data structure and loading data (see Figure 1). We also discuss how to extract raw data from structures used in Intel DAAL.

3.1. Data Structure

To organize and represent data, Intel DAAL uses objects referred to as “numeric tables”. There are multiple types of numeric tables for different structures of data.

- *Heterogeneous Tables* are used when there are multiple data types in a data set (e.g. `double`, `string`, etc.) There are two different structures supported: Structures of Arrays (SoA) and Arrays of Structures (AoS).

- *Homogeneous Tables* are used when the data set has only one type of data. There are two types of homogeneous tables: dense and sparse.

- *Matrices* are used when the application requires matrix algebra type workloads. There are three kinds of matrices supported: dense matrix, packed symmetric matrix, and packed triangular matrices.

Certain algorithms may have limitations on which numeric table it can work with. Refer to the DAAL documentation [3] for the specific algorithms to get information on which tables are supported.

The information about the contents, or the metadata, of numeric tables is stored in objects called data dictionaries. Data dictionaries are typically used for heterogeneous tables to keep track of various data types, but they can also be used for homogeneous tables to query information such as the number of rows in a table. Data dictionaries are often automatically created by DAAL, but in some cases, most notably AoS (Array of Structures) tables, they may have to be manually created.

Finally, the contents of the numeric table, the data, can be loaded from variety of locations. Intel DAAL has special interface objects called data sources, designed for loading data from some common data locations .

- `ODBCDataSource` - For data source with Open Database Connectivity (ODBC) API. Currently only supports MySQL.
- `FileDataSource` - For acquiring data from a source file. Currently only supports Comma Separated Values (CSV) file.
- `StringDataSource` - For interfacing with C-String format byte arrays.

3.2. Loading Data

In the remainder of the section, we will discuss data movement on homogeneous `NumericTables` through code samples. We first discuss two methods for loading data: populating from simple arrays and populating from a CSV file. Other modes will be discussed in part 2 and 3 of the series. Then we discuss how to recover raw data from numeric tables. For convenience, we work in these namespaces in the examples codes to follow:

```
1 using namespace std;
2 using namespace daal;
3 using namespace daal::data_management;
```

3.2.1. From simple arrays

This is perhaps the most versatile approach for populating a table, as it is generally simple to generate basic arrays from most sources of data (files, simulations, etc.) In this example we will show how to create a homogeneous table from an array of double precision values. Our discussion will only cover homogeneous tables; for heterogeneous data tables, see DAAL documentation [3].

The constructor for `NumericTable` allows for easy construction of a table from an array. The following code sample demonstrates how to create a double precision, 100×100 numeric table from an array.

```
// Array containing the data
const int nRows = 100;
const int nCols = 100;
double* rawData = (double*) malloc(sizeof(double)*nRows*nCols);

// Creating the numeric table
NumericTable* dataTable = new HomogenNumericTable<double>(rawData, nCols, nRows);

// Creating a SharedPtr table
services::SharedPtr<NumericTable> sharedNTable(dataTable);
```

Listing 1: Creating numeric table from basic arrays.

In the sample, we have gone a step further and created a `SharedPtr` table from the original `HomogenNumericTable` (see line 10). This is because many algorithms, including the linear regression algorithm, requires that the input data structures are of type `SharedPtr`.

3.2.2. From CSV files

CSV file format is one of the most widely used file storage formats. In a CSV file, individual data fields are separated by delimiters, which is conventionally a comma, and data points (entries) are separated

Intel DAAL allows us to directly import data from a CSV file using data source object of type `CSVFeatureManager`. Following example demonstrates how to load data from a CSV file using the feature manager.

```
string dataFileName = "/path/to/file/datafile.csv";
const int nRows = 1000; // number of rows to be read
// Create the data source
FileDataSource<CSVFeatureManager> dataSource(dataFileName,
                                   DataSource::doAllocateNumericTable,
                                   DataSource::doDictionaryFromContext);
// Load data from the CSV file
dataSource.loadDataBlock(nRows);

// Extract NumericTable
services::SharedPtr<NumericTable> sharedNTable;
sharedNTable = dataSource.getNumericTable();
```

Listing 2: Loading data from a CSV file.

Note that the CSV data source requires knowing how many rows are available in the CSV file to be read. In Listing 2 we have hard-coded the value 1000 for the number of rows, but this generally should be determined at runtime.

In Listing 2, we have used the options `doAllocateNumericTable` and `doDictionaryFromContext` when creating the data source. This is so that the `NumericTable`

and the `Dictionary` is created automatically from the contents of the CSV file. Although it is not covered here, data source objects also allow for manual allocation of `NumericTable` and `Dictionary` through `allocateNumericTableImpl()` and `setDictionary()` methods (see [3]).

If your data file has a separator (delimiter) that is not a comma, use the `setDelimiter()` method of the data source object. Note that this must be set *before* the data is loaded with `loadDataBlock()` method.

```
dataSource.setDelimiter(";");
```

3.3. Data Extraction

After populating a numeric table, you may need to extract raw data from it. There are two methods for getting the pointer to the raw data.

The most direct way is to simply use the `getArray()` method of the numeric table. Assuming that the table was populated with double precision values, the following code allows us to recover the raw data.

```
services::SharedPtr<NumericTable> dataTable;
// ... Populate dataTable ... //

double* rawData = dataTable.getArray();
```

Alternatively, the pointer to the data can be acquired by transferring the data to a `BlockDescriptor<TYPE>` object. The advantage of using the block object is that it allows one to get a subset of the data set by using `getBlockOfRows()` or `getBlockOfColumns()`.

```
services::SharedPtr<NumericTable> dataTable;
// ... Populate dataTable ... //

BlockDescriptor<double> block;
dataTable->getBlockOfRows(offset, numRows, readwrite, block);
double* rawData = block.getBlockPtr();
```

The arguments for `getBlockOfRows()` are as follows: `offset` is the index of the first row to load, `numRows` is the number of rows, `readwrite` determines the access model to the data (use DAAL presets, `readOnly`, `writeOnly` and `readWrite`), and finally, `block` is the `BlockDescriptor<TYPE>` object which the data is written into. After loading, the pointer to the array containing this data can be acquired by the `getBlockPtr()` method.

4. Example: Linear Regression

In our example, we apply a special case of linear regression algorithm, called polynomial regression (see Appendix A), to a single data set stored in CSV files. Linear regression requires three input data tables (discussed later in Section 4.1), so there are three CSV files, one for each data table. The data set contains all double-precision floating point values, and we only have a single compute node to do the regression.

Let us first analyze our situation in order to make decisions on the four choices presented in Figure 1. For algorithms, we will of course choose the linear regression algorithm. For the computation mode since all data is located in a single set of CSV files, and there is only one compute node, we will be using the batch computation mode. Since the data is stored in CSV files, we the data source object for CSV files to load data. Finally, as the data is all double-precision, we will use the homogeneous `NumericTable`.

4.1. How-to: Linear Regression

The basic workflow in linear regression consists of two steps;

- Training:
  In this step, the algorithm is "trained" with a training data set (a large number of feature sets, $\{\vec{x}_0, \vec{x}_1, ..., \vec{x}_{n-1}\}$, and the corresponding responses, $\{\vec{y}_0, \vec{y}_1, ..., \vec{y}_{n-1}\}$). The algorithm accumulates "knowledge" from analyzing this training set according to procedure described in Appendix A, and uses this for prediction in the next step.

- Prediction:
  Using the values learned from the training set, the algorithm takes an arbitrary set of test features, $\vec{x}_{test}$, and predicts the response variable $\vec{y}_{test}$ for this set of features. Typically, the larger the training set, the more accurate the prediction is.

The training part of linear regression requires two inputs: a numeric table containing sets of features and and a numeric table containing the corresponding responses for training. The output is what is referred to as "data model". This model contains the "knowledge" that was accumulated in the training phase.

The prediction part also requires two inputs: the data model from the training phase, and a numeric table containing sets of features to predict the response of. The output is the predictions of the responses to the test features sets.

Figure 4 illustrates the inputs required for the training and prediction.

On the training side in Figure 4, p is the number of the features, n is the number of data points for training, and k is the number of responses. For the prediction side, p and k are the same as training, and m is the number of data points for prediction. Note that n must be much greater than p for the linear regression algorithm to produce an accurate result. There is no limitation otherwise, however note that larger values of n, m, and p will all lead to a longer computation time.

In the remainder of the section, we demonstrate how to implement the training and prediction parts of linear regression using code samples.

In the code samples to follow, we will be working in these namespaces:

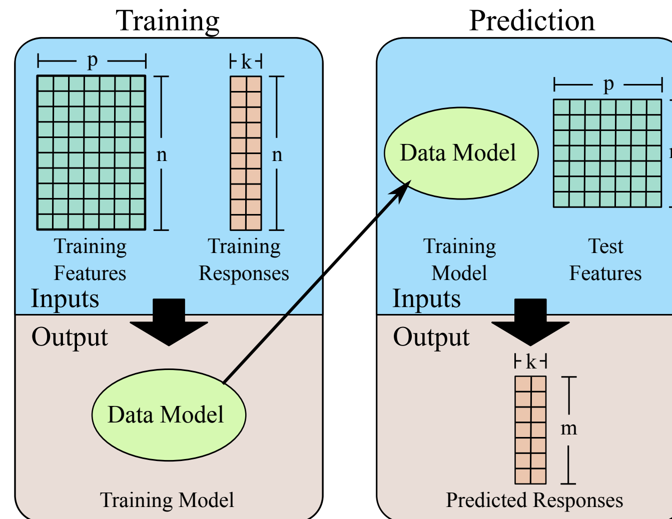

Figure 4: The inputs and output of the training and prediction .

```
using namespace daal;
using namespace daal::data_management;
using namespace daal::algorithms::linear_regression;
```

4.1.1. Batch Training

The general workflow for batch training is as follows: create an algorithm object, train the algorithm with the training sets, then extract the trained model. Listing 3 demonstrates the training phase of the linear regression algorithm in the batch mode.

```
// Setting up the training sets.
services::SharedPtr<NumericTable> trnFeatures(trnFeatNumTable);
services::SharedPtr<NumericTable> trnResponse(trnRespNumTable);

// Setting up the algorithm object
training::Batch<> algorithm;
algorithm.input.set(training::data, trnFeatures);
algorithm.input.set(training::dependentVariables, trnResponse);

// Training
algorithm.compute();

// Extracting the result
services::SharedPtr<training::Result> trainingResult;
trainingResult = algorithm.getResult();
```

Listing 3: Training the linear regression algorithm in batch mode.

First, the features and the responses for the training set must be created (line 1-3 in Listing 3). For

more details on creating data tables, refer to Section 3.2.

The next step is to create an algorithm object. In our example, we have created the algorithm using default settings by having empty `Batch<>`, but the options can be set inside the brackets. With linear regression there are two options that can be set:

```
training::Batch<algorithmFPType=TYPE, method=MTHD> algorithm;
```

- `algorithmFPType` – Floating-point precision to be used for the computation by the algorithm. `TYPE` can be either `float` or `double`. By default this is set to `double`.
- `method` – The mathematical algorithm to be used for computation. For linear regression, `MTHD` can be `defaultDense` or `qrDense`. By default this is set to `defaultDense`.

After creating the algorithm object, the inputs of the algorithm must be set (lines 6-8). The linear regression algorithms require two inputs, `training::data` and `training::dependentVariables`, which are the training features and the training responses respectively. The inputs are set by using `input.set()` method of the algorithm object. Then to train the algorithm with this data set, use the `compute()` method of the algorithm object. Recall that with batch mode only supports one data set. If this procedure, setting the input data set and using the `compute()` method, is repeated with different data sets but with the same algorithm object, then the result will only reflect the last data set. To train with multiple data sets, distributed or online mode should be used. These modes will be discussed in the 2nd and the 3rd papers.

Finally, to extract the result, use `getResult()` method of the algorithm object, which returns a `training::Result` object. This result object contains the "knowledge" that algorithm acquired (i.e. the coefficients of linear regression), and will be used in the prediction phase.

4.1.2. Prediction

For the prediction phase, only the batch mode is supported. Listing 4 demonstrates the prediction phase.

```
// ... Set up: training algorithm (see Section 3.2)  ... //
services::SharedPtr<training::Result> trainingResult;
trainingResult = algorithm.getResult();
services::SharedPtr<NumericTable> testFeatures(tstFeatNumTable);
// ... Set up: populating testFeatures (see Section 2) ... //

// Creating the algorithm object
prediction::Batch<> algorithm;
algorithm.input.set(prediction::data, testFeatures);
algorithm.input.set(prediction::model, trainingResult->get(training::model));

// Training
algorithm.compute();

// Extracting the result
services::SharedPtr<prediction::Result> predictionResult;
predictionResult = algorithm.getResult();
BlockDescriptor<double> resultBlock;
predictionResult->get(prediction::prediction)->getBlockOfRows(0, numDepVariables,
                                                               readOnly, resultBlock);
double* result = resultBlock.getBlockPtr();
```

Listing 4: Prediction using the computed result

Before the prediction stage, the training result must be acquired (see Section 4.1.1) and the test features table must be created (see Section 3.2).

For prediction, we will be using a `prediction::Batch<>` object. Note that this is different from the `training::Batch<>` object we used in Section 4.1.1. The prediction algorithm has two input options `algorithmFPType` and `method`, which behave the same way as the options for training object (see 4.1.1).

Just as the training objects, the inputs of the algorithm must be set using the `input.set()` method. The prediction algorithm require two inputs, `prediction::data` and `prediction::model`, which are the test features and the model from training. The model can be acquired from the training result by invoking the method `get(training::model)`. After the inputs are set, use the `compute()` method of the training object to get the prediction of responses to the test feature vectors.

To get the predicted values, first extract the result object from the algorithm using the `getResult()` method. The result object contains a `NumericTable` with the result of the prediction, from which the pointer to the resultant array can be recovered. For more on extracting data from numeric tables, refer to Section 3.3.

The i-th element in every row of the prediction array corresponds to the predicted result for the i-th test feature vector. If there are multiple responses, then, remember that the `getBlockPtr()` returns a one-dimensional result; the prediction for j-th response (out of `numResponses`) to the i-th test feature vector is in `result[i*numResponses + j]`.

4.2. Example implementation

The source file batch_polynomial_regression.cc implements polynomial regression using batch mode. Polynomial regression is a special case of linear regression that fits the response to a polynomial function of the features (see Appendix A). In this sample, we assume a workload where all data is stored in one set of three CSV files on one compute node (see Section 4). For information on batch mode and loading from CSV files, refer to 4.1.1 and 3.2.2 respectively.

To implement polynomial regression, the features must be expanded to a polynomial (e.g $\{x_1, x_2, ..., x_n\} \rightarrow \{x_1, x_1^2, x_1^3, ..., x_1^e, x_2, x_2^2, x_2^3, ..., x_n^e\}$ for some expansion order, e). Thus, we must recover the data from the numeric table acquired from the CSV data source, expand it to multiple orders, then repackage the expanded features into a numeric table. Listing 5 demonstrates this procedure.

```
1  // Getting data from the source
2  FileDataSource<CSVFeatureManager> featuresSrc(trainingFeaturesFile,
3                                                DataSource::doAllocateNumericTable,
4                                                DataSource::doDictionaryFromContext);
5  featuresSrc.loadDataBlock(nTrnVectors);
6  // Creating a block object to extract data
7  BlockDescriptor<double> features_block;
8  featuresSrc.getNumericTable()->getBlockOfRows(0, nTrnVectors,
9                                                 readOnly, features_block);
10 // Getting the pointer to the data
11 double* featuresArray = features_block.getBlockPtr();
12 // Expanding the data (see source for full implementation)
13 const int features_count = nFeatures*expansion*nTrnVectors;
14 double * expanded_tstFeatures = (double*) malloc(sizeof(double)*features_count);
15 expand_feature_vector(trnFeatures_block.getBlockPtr(), expanded_trnFeatures,
16                                           nFeatures, nTrnVectors, expansion);
17 // Repackaging the result into a numeric table
18 HomogenNumericTable<double> expanded_table(expanded_trnFeatures,
19                                             nFeatures*expansion, nTrnVectors);
20 trainingFeaturesTable = services::SharedPtr<NumericTable>(expanded_table);
```

Listing 5: Expanding the features to a polynomial of order set by expansion.

Listing 5 only shows the expansion process for training features, however, both the training and test feature vectors must be expanded. After the expanded numeric table is populated, the rest of the implementation is nearly identical to the example shown in Section 4, so it will not be reiterated here.

The final working source code, batch_polynomial_regression.cc was designed to be modular so that an user can input any (valid) set of the three input CSV files and perform linear regression on the data set. For instruction on how to compile and run the example, refer to the included README file.

The code, batch_polynomial_regression.cc can be found at the Colfax Research website [1].

Actual	1st Order (error)	2nd Order (error)	3rd Order (error)
3.85	11.85 (+8.00)	2.82 (-1.03)	1.68 (-2.17)
8.04	17.46 (+9.42)	11.84 (3.80)	7.54 (-0.50)
7.95	18.67 (+10.72)	13.51 (5.56)	9.24 (1.29)
32.75	27.15 (-5.60)	34.40 (1.65)	34.97 (2.22)
33.97	27.17 (-6.80)	34.10 (0.13)	34.85 (0.88)
0.09	-10.42 (-10.51)	4.97 (4.88)	-1.86 (-1.95)
1.97	5.96 (+3.99)	-2.45 (-4.42)	0.65 (-1.32)
RSME	±8.21	±3.63	±1.60

Table 1: Reference result vs prediction for 1st, 2nd and 3rd order expansions. RSME is the root-mean-squared-error.

4.3. Results

The resultant example code was tested using a real-life data set; hydrodynamics of yachts. The features of the data set are 5 parameters of the hull geometry of the yacht and a value called Froude number. The response we want to predict is the residual resistance, which is a measure of how difficult it is to push the yacht through water. Ability to determine this resistance value from features is extremely useful for yacht designers, because it gives them information about the engine requirements of the yacht at the design stage. For more information on the data set, refer to [4] and [5].

This data set was downloaded from the UCI Macine Learning Repository [6]. The data set has p=6 features and k=1 responses in 307 data points. The details of the data set can be found on the landing page for this data set in UCI Machine Learning Repository [7]. The features, as listed on the page, are;

1. Longitudinal position of the center of buoyancy, adimentional.
2. Prismatic coefficient, adimentional.
3. Length-displacement ratio, adimentional.
4. Beam-draught ratio, adimentional.
5. Length-beam ratio, adimentional.
6. Froude number, adimentional.

To test our application, we split this data set of 307 data points. 7 data points are randomly selected to be used as the test data set, and remaining 300 data points are used for training our model.

The algorithm was trained with the 1st, 2nd and 3rd order expansions of the features. Unfortunately, the data set could not support 4th order expansion, and the DAAL linear regression algorithm exited with an error at that expansion. Table 1 compares the prediction form the three data models to the actual reference result. As evident from the root-mean-squared-error (RMSE) the data model produced from the training becomes increasingly more accurate as we expand the features to 2nd and 3rd orders.

Figure 5 illustrates another test of the accuracy of the model. We generated a test feature set by keeping $x_1, ..., x_5$ constant and varying x_6, then used the model to predict the responses for this test data set. Then we selected 14 data points from the real data set that have $x_1, ..., x_5$ that matches our test set, and compared

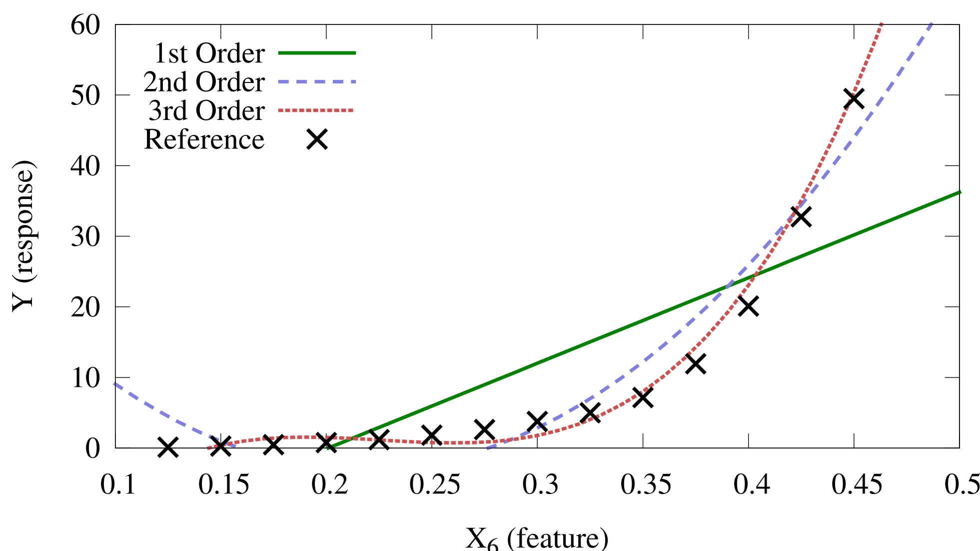

Figure 5: The predictions with 5 features (x_1 to x_5) held constant. The "Reference" values are actual values obtained from the original data set.

the results. Just as Table 1, Figure 5 also shows the improvement in the accuracy of the model with the higher order expansions.

5. Closing words

This publication was the first part in a series of three white papers introducing Intel DAAL. In this part, we discussed how to implement a basic linear regression application using Intel DAAL. In the next two parts, we will cover more advanced topics. In part 2, we discuss distributed computation of linear regression algorithm using the distributed mode and MPI. In part 3 we discuss the online mode and streaming data from a MySQL database. The source code for this series, as well as part 2 and 3 of the series, can be found at the Colfax Research website [1].

References

[1] Ryo Asai. Introduction to Intel DAAL, Part 1 of 3: Polynomial Regression with Batch Mode Computation, 2015 *(landing page for this paper)*.
http://colfaxresearch.com/intro-to-daal-1/.

[2] James Reinders. Intel Data Analytics Acceleration Library.
https://software.intel.com/en-us/daal.

[3] Download link for DAAL User and Reference Guide.
https://software.intel.com/en-us/intel-daal-support/documentation.

[4] R. Onnink J. Gerritsma and A. Versluis. Geometry, resistance and stability of the delft systematic yacht hull series. *International Shipbuilding Progress*, 28:276–297, 1981.

[5] R. Lopez I. Ortigosa and J. Garcia. A neural networks approach to residuary resistance of sailing yachts prediction. *In Proceedings of the International Conference on Marine Engineering MARINE*, 2007.

[6] M. Lichman. UCI machine learning repository, 2013.
http://archive.ics.uci.edu/ml.

[7] M. Lichman. Landing page for yacht hydrodynamics data in UCI Machine Learning Repository.
http://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics.

Appendix A. Mathematics of regression

The goal of regression is as follows: given a set of independent variables $\vec{x}$ (called "feature vector" or "features") and a dependent variable y (called "response"), predict the value of y given any arbitrary $\vec{x}$. For example, regression can be used to predict the crop-yield of a farm (our dependent variable y), given the size of the farmland and amount of fertilizer used (our independent vector $\vec{x}$).

In the rest of the section, we will look at the mathematical background of linear regression and polynomial regression.

A.1. Linear regression

In its most basic form, linear regression is simply solving the equation;

$$y = bx + b_0 \tag{1}$$

In this case, to solve for the slope b and the intercept b_0 the minimum required number of dependent-independent variables pair (let us call it n) is 2. Solving for these variables is the training phase. After determining the values b and b_0 from the training data set, our algorithm can predict the response y from any arbitrary feature x. This is the prediction phase. Of course, this is an over-simplification. In majority of cases, there is some error (let us call it δ) that is inherent to the problem, so the equation becomes.

$$y = bx + b_0 \pm \delta \tag{2}$$

With the error introduced, we need a larger training data set of x and y to determine the value of b and b_0. This type of problem is referred to as single linear regression, or simply linear regression.

However in real-life applications, it is often the case that a response depends on multiple features. If there are p features that the response y depends on, the equation becomes;

$$y = b_0 + \sum_{i=1}^{p} b_i x_i \pm \delta \tag{3}$$

The goal in such a problem is to determine the values $b_0, b_1 ... b_p$. This workload is called multiple linear regression. To solve such a problem, we need multiple responses y, produced from different feature vectors $\vec{x}$. Let's assume that we have n such data points, and we will call this the training set.

Determining the values of $b_0, b_1 ... b_p$ with the training set is similar to solving a system of equations. We can represent this workload as;

$$\vec{y} = \mathbf{A} \cdot \vec{b} \pm \delta \tag{4}$$

Here, $\vec{y}$ is the response vector of size n, and $\vec{b} = \{b_0, b_1 ... b_p\}$ is a vector of size $p + 1$ that we are solving for. Note that we have included the intercept, b_0 in $\vec{b}$ to adhere to the convention. $\mathbf{X}$ is a $n \times (p+1)$ matrix; the first column is all 1 for the b_0 and the following p columns represents the p features.

There are several methods for determining the values of $\vec{b}$ that will most accurately approximate the data set (e.g. "least squares optimization"). Implementation of these methods are beyond the scope of this paper.

In theory, the number of independent variables p can be any arbitrarily high number. But the larger the p, the larger the required training set (e.g. greater n) for an accurate prediction.

A.2. Polynomial regression

In many real applications, the response may not change linearly with a given feature. To make more accurate predictions in such applications, a special case of linear regression called polynomial regression can be used. In polynomial regression, we assume that the response changes as a polynomial function of a give n feature. In the case of one feature, x, the problem becomes.

$$y = (\sum_{i=0}^{\infty} b_i x^i) \pm \delta \tag{5}$$

Of course, we cannot compute to the upper limit of ∞. Thus, a finite upper-bound, call it e, has to be picked. Also, as in the case of linear regression, a single feature may not be sufficient. We can extend this regression to p features. The final equation for $b_{i,j}$ becomes:

$$y = (\sum_{j=1}^{n} \sum_{i=0}^{e} b_{i,j} x_n^i) \pm \delta \tag{6}$$

At first glance this may appear much more complex than the linear regression case, however this is in fact equivalent to equation (3). The only difference with the linear case is that we have powers of x in the equation, but recall that x is the feature (i.e., this is the input). Given the set of p features and the expansion e, we can construct a single feature vector $\vec{x}$ of length $e \times p$ such that $\vec{x} = \{x_1, x_1^2, x_1^3, ...x_1^e, x_2, x_2^2, x_2^3, ..., x_p^e\}$. Substituting this in and collapsing the summations, the equation becomes;

$$y = b_0 + (\sum_{i=1}^{e*p} b_i x_i) \pm \delta \tag{7}$$

Which is identical to equation (3). (Note that all constants $b_{0,j}$ got combined into a single b_0)

Just like in linear regression, in theory $e \times p$ can be arbitrarily high. But once again, large value of the product $e \times p$ will require a correspondingly large number of training data points, n, for an accurate prediction.