LibFeature: A software library for quickly generating feature vectors on the fly from structured data

Dominic Mazzoni

Jet Propulsion Laboratory, California Institute of Technology,
Pasadena, CA 91109, USA
Dominic.Mazzoni@jpl.nasa.gov

Abstract. We have developed a software library, LibFeature, that greatly simplifies the task of extracting feature vectors from raw data. The instructions for computing feature vectors from the input data are written in a high-level language, which can be interpreted in real-time, but because the language is deterministic, it can be executed on many feature vectors in parallel, resulting in performance comparable to efficient C code. We describe the capabilities of LibFeature, the Feature Description Language (FDL), the internal architecture and optimizations, and then show some benchmarks of its performance, while using realistic examples of constructing features from scientific image and time-series data throughout.

1 Introduction

When applying a machine learning or data mining algorithm to a new data set, it is often the case that the majority of the researcher's time is spent writing code to parse the data set and manipulate it into a form that can be used by the algorithm. Many such algorithms expect data to be in the form of *feature vectors*, each of which contains all of the information that the algorithm has available about one input example. For scientific and engineering data sets, feature vectors are typically composed of all real numbers representing measurements such as the image intensity at a particular pixel, or the electrical current of an instrument at a particular time. Assuming that these values are already available in data files of a known format, various operations might need to be done in order to construct feature vectors from the raw data, including:

1. Combining data from multiple files or multiple tables, for example if each band of a multispectral imager is stored in a separate array
2. Changing the units or normalizing values so that all elements in each feature vector are approximately the same magnitude
3. Computing features based on mathematical functions of raw values, for example computing the Normalized Difference Vegetation Index (NDVI) given radiance at both red and near-infrared wavelengths
4. Converting from categorical features to numerical, for example a sensor that reads `OFF`, `READY`, or `ERROR` could be represented by the features 0, 1, and 2, or alternatively by the vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$
5. Augmenting the feature vector for one particular time or location with features from neighboring examples, for context
6. Filtering out feature vectors when one of the features is missing or bogus, or alternatively, interpolating the value of missing features in that case

We have written a software library, called *LibFeature*, that attempts to make this process easier by allowing one to specify the commands to produce a feature vector in a high-level language. Because LibFeature is highly optimized and because of the inherent parallelizability of the problem, it is usually possible to use LibFeature without suffering any speed penalty for using the additional layer and the parsing of the high-level language. In fact, LibFeature is often within a factor of two of a straightforward C implementation, and thus it is fast enough to be used in near-real-time systems.

One of the major challenges in designing a software library to replace a programming task that is time-consuming but not difficult, is that the new software library must be especially clean, lightweight, portable, and easy to use, otherwise most people will find it easier to just reinvent the wheel each time rather than introduce a dependency on another library. LibFeature is written in very portable C and is designed to compile and run on almost any modern

computing platform, including Windows, Mac OS X, Linux, and any modern Unix system. It is powerful enough to handle surprisingly complicated calculations, but the most common tasks are designed to be as easy as possible.

In this paper, we will describe an outline of the system and the major capabilities and limitations, introduce the feature description language, show some examples of many common tasks that can be accomplished using LibFeature, discuss how LibFeature works and some of the optimizations it uses in order to achieve good performance, and finally examine some benchmarks comparing its performance to C programs.

2 Related Work

While we are not aware of any previous work to develop a software library specifically for constructing feature vectors for machine learning and data mining use, nevertheless many other software programs and systems served as inspiration or guidance for various aspects of LibFeature.

The idea of constructing a virtual matrix containing all of the results and then accessing only individual rows (feature vectors) as needed was inspired by the typical use of Structured Query Language (SQL) [1] within a high-level programming language. A program accessing an SQL database will typically use one function to execute an SQL query, which returns a handle to a result set (e.g., SQLExecute in the standard SQL Call Level Interface (SQL/CLI)). The program then calls a separate function to access individual rows from the result set (e.g., SQLFetch in SQL/CLI).

The idea of compiling a high-level language into an intermediate bytecode which can then be executed more quickly was inspired primarily by Java's bytecode [2] [3], which popularized the concept. Finally, the idea of using a somewhat restricted high-level language to specify instructions that are executed quickly on thousands of vectors in parallel is very similar to pixel shading languages such as the OpenGL Shading Language [4], which allows programmers to write simple programs that are executed directly on a graphics card to determine the final color of each pixel in a 3-D rendered image.

3 System Outline

LibFeature is initialized with one or more input arrays, and a Feature Description Language (FDL) program specifying how to create the feature vectors. On initialization, LibFeature parses the program but does not compute any actual feature vectors, but instead returns a *virtual matrix* of feature vectors. For example, if you have a 50×50 image of pixels and each feature vector has 7 features, the virtual matrix would be of size $2,500 \times 7$. (By convention, we assume that the matrix is stored in *row-major* order, in which case each feature vector is one row. It is equally correct, however, to imagine a column-major matrix where each feature vector is a column; this may be preferable to Fortran and Matlab users, and in either case the memory representation is the same.)

From a high-level language, you then query LibFeature for a single row of the virtual matrix (one feature vector), a set of contiguous rows, or an arbitrary list of rows. LibFeature is designed to work with very large virtual matrices that could never be computed and stored in memory all at once; specifically there is no limitation that the number of total elements in the matrix fit in one 32-bit integer. It is quite common to initialize a virtual matrix with ten trillion elements, but only actually compute a few thousand rows randomly scattered throughout the matrix.

Internally, LibFeature works with only floating-point numbers. While input arrays are allowed to be any numerical data type, everything is converted to (your choice of single- or double-precision) floating-point values first in the resulting feature vector. This was a reasonable simplifying design decision because on modern CPUs there is little performance difference between floating-point and integer calculations.

3.1 Input Arrays and Dimensions

LibFeature can read from arrays in memory, or directly from binary files on disk. When reading from disk, you can specify the data type, start offset, endianness, and dimensions, or LibFeature can determine it automatically from the file's header (supported file types currently include BMP, Matlab, NetCDF, PGM, PPM, and TIFF). (LibFeature can be used with numerical data stored in ASCII text files, too, but currently this requires reading the entire file into memory first, so it does not scale as well.) LibFeature works with arbitrary multidimensional arrays, but it requires that you specify which dimensions measure unique data points (extrinsic dimensions), and which

dimensions measure different fields within the same data point (intrinsic dimensions). For example, consider a typical color image file with dimensions of 640×480 pixels, and three color components (red, green, and blue) per pixel. The total length of the file is $640 \times 480 \times 3$ elements, however only 640×480 of these are unique data points. The last dimension is an intrinsic dimension, because it counts the number of elements within each pixel (in this case, 3). LibFeature uses the convention of using positive numbers to denote extrinsic dimensions, and negative numbers to denote intrinsic dimensions, given in the order of fastest-moving to slowest-moving.

Most color 640×480 images, then, would have dimensions $(-3, 640, 480)$ because the file is stored like this:

$$480 \left\{ \begin{array}{l} \overbrace{\text{RGB RGB RGB} \ldots \text{ RGB}}^{640} \\ \text{RGB RGB RGB} \ldots \text{ RGB} \\ \vdots \qquad\qquad\qquad \ddots \quad \vdots \\ \text{RGB RGB RGB} \ldots \text{ RGB} \end{array} \right.$$

Users of scientific image datasets may be used to calling this the Band-Interleaved-by-Pixel (BIP) format. However, some image files store the red, green, and blue channels as separate planes, meaning that the first 640×480 elements in the file are for the red channel, which is followed by all of the green channel, and then all of the blue. Scientific users would call this a Band-Sequential (BSQ) file. In this case, the dimensions that you pass to LibFeature would be $(640, 480, -3)$:

$$\begin{array}{l} 480 \left\{ \begin{array}{l} \overbrace{\text{R R R} \ldots \text{ R}}^{640} \\ \vdots \qquad \ddots \ \vdots \\ \text{R R R} \ldots \text{ R} \end{array} \right. \\ 480 \left\{ \begin{array}{l} \text{G G G} \ldots \text{ G} \\ \vdots \qquad \ddots \ \vdots \\ \text{G G G} \ldots \text{ G} \end{array} \right. \\ 480 \left\{ \begin{array}{l} \text{B B B} \ldots \text{ B} \\ \vdots \qquad \ddots \ \vdots \\ \text{B B B} \ldots \text{ B} \end{array} \right. \end{array}$$

The default assumption in LibFeature is that you want to generate exactly one feature vector per data point. In section 4.1, we will see how to exclude some data points from getting turned into feature vectors, and in section 4.2, we will see how to generate multiple feature vectors from one data point. But most of the time there is a direct correspondence. The concept of extrinsic and intrinsic dimensions and LibFeature's convention of using negative indices is a surprisingly powerful way to represent mappings between the input array and feature vectors. As a more complicated example, suppose that you want to take the same image above, but you only want 320×240 feature vectors - one for each 2×2 group of pixels. All that is required is to tell LibFeature that the dimensions of the file are $(-3, -2, 320, -2, 240)$. LibFeature can infer from this that there will be 320×240 feature vectors, and makes it effortless for you to combine elements from the four pixels in each 2×2 group into each feature vector. Another more common illustration of the power of this convention is that when using LibFeature, switching from a BSQ to a BIP input file often requires changing only one line of code.

3.2 Feature Selection

A common task in machine learning problems is to start with a large feature set containing dozens or hundreds of potential features, and use a feature selection algorithm to choose a subset of features out of these. LibFeature is designed to make this quite easy and efficient. At any time, you can pass LibFeature a *mask array*, specifying which of the columns of the virtual matrix, corresponding to features, you are interested in. Not only will LibFeature take this into account, returning only the columns of the virtual matrix of interest from then on, but it also quickly re-optimizes its computation to avoid unneeded computations. Thus, there is no penalty for writing an FDL program to compute every possible feature you could imagine, and then later selecting the actual features you want to use empirically.

4 FDL: The Feature Description Language

The Feature Description Language (FDL) is used to specify how each feature vector is constructed from the input arrays. The syntax is based on Lisp-like S-expressions, and while it contains many capabilities commonly found in programming languages (including many built-in mathematical functions, conditionals, user-defined functions, and macros) it is an imperative language and purposefully does not include any flow control. The reason for the lack of

flow control is that the same commands are applied to thousands of feature vectors in parallel, allowing for substantial optimizations.

Recall that S-expressions use *prefix* notation, where the name of the function or operator is always the first element of a parenthesized list, and the arguments follow. For example, the formula to compute the approximate area of a circle given its radius r could be expressed as:

```
(* 3.14159 (* r r))
```

An FDL program produces one feature vector given one particular input data point. The command to produce a feature is `output`. To retrieve the value of an input data point (pixel, in the case of an image), simply use the name of the input array (which was assigned during initialization) as if it was a function. Supposing that our RGB ($-3 \times 640 \times 480$) image was named `myimage`, then the simplest possible FDL program would be:

```
(output (myimage))
```

This program would only result in one feature per pixel. To output three features, one each for red, green, and blue, pass an argument to `myimage` indicating the offset of the element you want:

```
(output (myimage 0))
(output (myimage 1))
(output (myimage 2))
```

A 0 is always implied if there is no argument. Note that the offset can appear in any dimension, not just the first dimension. So it would be very easy to make each feature vector contain six features - the RGB values of the current pixel, and the RGB values of the pixel above and to the left of that one:

```
(output (myimage  0  0  0))
(output (myimage  1  0  0))
(output (myimage  2  0  0))
(output (myimage  0 -1 -1))
(output (myimage  1 -1 -1))
(output (myimage  2 -1 -1))
```

Note that the order of the offsets corresponds to the order of the dimensions: the first offset is relative to the fastest-moving dimension, and so on. Finally, note that when an offset takes you out of the bounds of an image, LibFeature replaces those features with NaN (not-a-number). It is easy to simply eliminate any feature vectors that have any NaNs in them later.

FDL lets you create variables using the `set` command, and it also comes with most standard mathematical functions. Here's a more complicated example, then, that outputs the three features per pixel, but normalizes them so that they always sum to 1:

```
(set red   (myimage 0))
(set green (myimage 1))
(set blue  (myimage 2))
(set sum (+ red green blue))
(output (/ red sum))
(output (/ green sum))
(output (/ blue sum))
```

This works as expected, unless the red, green, and blue values happen to all be 0. LibFeature will properly return NaNs when this happens, but if you would prefer to output zeros instead, use a conditional: `(if` *expr true-value false-value*`)`

```
(set red   (myimage 0))
(set green (myimage 1))
(set blue  (myimage 2))
(set sum (+ red green blue))
(output (if (== sum 0) 0 (/ red sum)))
(output (if (== sum 0) 0 (/ green sum)))
(output (if (== sum 0) 0 (/ blue sum)))
```

Note that like C, Java, Perl, etc., Libfeature uses a double-equals (`==`) to test for equality.

As a final example, here is an FDL program that averages the pixel values over a 3×3 region. This will introduce a `for` loop. Even though FDL does not have any loops that execute a different number of times depending on the status of real-time calculations, it does include constructs that allow loops that execute a fixed number of times. There are more variations, but the simplest syntax for a `for` loop is `(for` *variable* `(seq` *start stop*`)` *command* `[`*commands...*`])`, where `seq` is a built-in function that returns a list of elements to be iterated over, from *start* to *stop*, inclusive. Here's the code:

```
(set red 0)
(set green 0)
(set blue 0)
(for i (seq -1 1)
  (for j (seq -1 1)
    (set red (+ red (myimage 0 i j)))
    (set green (+ green (myimage 1 i j)))
    (set blue (+ blue (myimage 2 i j)))))
(output (/ red 9))
(output (/ green 9))
(output (/ blue 9))
```

4.1 Validation

In the most recent example FDL program above, one problem is that for all of the pixels along the edges of the image, the 3×3 neighborhood extends outside the image bounds, resulting in NaNs. One way to solve this would be to change the FDL code so that it only sums the values if they are not NaN. However, sometimes it might make more sense to simply exclude those pixels from becoming feature vectors. So for our 640×480 image, we would only end up with 638×478 feature vectors. In LibFeature this can be done using the `require` command. When the argument to `require` evaluates to false, the entire feature vector is marked as invalid. (Then at runtime, the user can choose to extract all feature vectors in a given range, or only the valid feature vectors in that range.) To test if a value is not NaN, you can use the `finite` function. Here is the above program with validation:

```
(set red 0)
(set green 0)
(set blue 0)
(for i (seq -1 1)
  (for j (seq -1 1)
    (set red (+ red (myimage 0 i j)))
    (set green (+ green (myimage 1 i j)))
    (set blue (+ blue (myimage 2 i j)))))
(require (finite red))
(require (finite green))
(require (finite blue))
(output (/ red 9))
(output (/ green 9))
(output (/ blue 9))
```

Note that it would have worked equally well to put the `require` inside of the for loop. When there are a lot of feature vectors that will be marked as invalid, it is usually fastest to do the validation as early as possible, because LibFeature can stop computing the feature vector as soon as it is marked invalid. When most vectors will be valid, it's best to execute the `require` command as few times as possible.

4.2 Variations

One way to cut down on overfitting in machine learning problems is to *jitter* the input, turning each input point into multiple feature vectors, each one offset by a small amount. LibFeature makes it easy to have one input point result in multiple feature vectors. Simply pass an initial argument to the `output` and `require` commands indicating the 0-based index of the feature vector to be output. Each different feature vector is then called one *variation*. Here's a simple FDL program that outputs two feature vectors per pixel: one vector is the original pixel, and the other is a darker version of the same pixel:

```
(set red   (myimage 0))
(set green (myimage 1))
(set blue  (myimage 2))
(output 0 red)
(output 0 green)
(output 0 blue)
(output 1 (* 0.9 red))
(output 1 (* 0.9 green))
(output 1 (* 0.9 blue))
```

4.3 Multiple input arrays

In all of the examples above, we have been assuming that LibFeature took just a single input array. However, LibFeature was designed to work with multiple inputs. There are two common ways that multiple inputs are used. The first way is when the data for a single image or other dataset is already stored in multiple files. For example, a color image might be stored as three separate files named `myimage.red`, `myimage.green`, and `myimage.blue`. In this case, you would initialize LibFeature with all three arrays and give them different names. Since the dimensions of all of the files are the same, LibFeature would automatically align them and generate only one set of feature vectors. The second way that multiple inputs can be used is when you want to concatenate feature vectors from multiple, independent files. If you initialize LibFeature with multiple input files but give them all the same name, LibFeature will process feature vectors from the files consecutively. So if you had two images, one 640×480 and one 800×600, LibFeature would return a virtual matrix with $640 \cdot 480 + 800 \cdot 600 = 787,200$ rows. This is particularly useful for training machine learning algorithms on a random subset of vectors. Rather than computing feature vectors for every pixel in hundreds of images, simply initialize LibFeature with all of the images, get one virtual matrix, and then retrieve random rows from that matrix until training has converged.

4.4 Labels

Training a supervised classifier requires a class label for every feature vector. While it would be possible to simply denote the last element of each feature vector as the class label, in practice it is often convenient to have the labels available separately. LibFeature provides a special mechanism to output a single label associated with each vector, using the `label_output` command.

5 Limitations and workarounds

While FDL is powerful enough to do many computations and generate surprisingly complex feature vectors (see section 7.2), there are certainly several types of features that FDL is not adept at expressing. In particular, FDL assumes that each feature vector can be generated independently and deterministically. As a simple example, there is no way in FDL to specify that a particular feature is to be normalized, because this would require scanning through the entire input first. One solution is to use LibFeature in two passes: In the first pass, LibFeature can be used to scan the input files and return vectors containing the values to be normalized. The program can then compute the normalization coefficients, and call LibFeature a second time with these coefficients in the FDL. As another example, it is common to compute the wavelet decomposition of an image to derive texture features, but FDL is not designed for transformations that operate on an entire image at once. In this case, it is best to use another program to preprocess input images and generate transformed images. LibFeature can still be useful for generating feature vectors from these transformed inputs, though, to combine the untransformed and transformed inputs with different weights.

6 Inside LibFeature

In this section, we will examine how FDL programs are converted to an internal representation and executed to produce the feature vectors. Readers who are only interested in using LibFeature as a black box may want to skip directly to section 7 or 8. So as to illustrate the diversity of problems for which LibFeature is applicable, we will switch from using an image data set as the main example to a time-series analysis problem. Suppose that your data contains several million readings taken at periodic intervals from some scientific instrument. To make things more interesting, also suppose that the data is somewhat noisy, and you want to smooth it out using a very simple convolution filter: every point will be replaced with the mean of the values within a certain neighborhood centered at that point. Here's an FDL program that computes a feature vector consisting of each value and the mean of its temporal neighborhood, supposing that the input file is named `sensor_input`:

```
(set nborhood 2)
(set len (+ 1 (* 2 nborhood)))
(for i (seq (- 0 nborhood) nborhood)
  (append a (sensor_input i)))
(set psum (+ a))
(output 0 (sensor_input 0))
(output 0 (/ psum len))
```

When LibFeature parses this FDL program, it interprets each S-expression in order, building up its internal data structures. Every expression it encounters turns into a command, where some commands use references to previous commands as parameters. Note that `set` does not result in a command, but instead adds an entry to a symbol table, allowing you to refer to the result of a particular expression by name. Here is a representation of the commands that are generated when the FDL program above is parsed:

```
v[0] = 2;
v[1] = 5;
v[2] = array("sensor_input", -2);
v[3] = array("sensor_input", -1);
v[4] = array("sensor_input", 0);
v[5] = array("sensor_input", 1);
v[6] = array("sensor_input", 2);
v[7] = sum(v[2], v[3], v[4], v[5], v[6]);
v[8] = array("sensor_input", 0);
v[9] = v[7] / v[1];
```

This representation is meant to give you an idea of how the particular feature vector described above could be computed using a temporary array `v[]`. The array `v[]` is analogous to register on a processor. Internally, LibFeature represents these commands using a bytecode, but the information contained is the same. In this particular case, after the nine commands are finished for each input point, the outputs are in `v[8]` and `v[9]`.

6.1 Optimizations

Note that there is a little bit of redundancy in the sequence of commands shown above. The first value,

v[0]=2 (which came from nborhood) is never actually used. Also, v[4] and v[8] are identical. LibFeature performs some optimizations on its command sequence before executing it. These optimizations are very similar to optimizations performed by modern compilers, but because there is no flow control, they are all relatively straightforward, and in fact all of the optimizations run in time $O(n \log n)$ (where n is the number of commands). The optimizations performed by LibFeature include:

1. Constant Propagation: Expressions that return constant values are replaced with the result of that expression. This was already seen in the above command sequence, because the expression (+ 1 (* 2 nborhood)) was replaced with the result, 5, in a single step.
2. Identity Reduction: Computations that involve multiplying by one or adding zero are eliminated. This encourages users to write code with lots of tunable parameters, since an additive parameter can simply be set to 0 to eliminate it.
3. Common Subexpression Elimination: Commands that are duplicates of earlier commands are eliminated.
4. Dead Code Elimination: Commands which are never used are eliminated.

After optimization, the command sequence looks like this:

```
v[0] = 5;
v[1] = array("sensor_input", -2);
v[2] = array("sensor_input", -1);
v[3] = array("sensor_input");
v[4] = array("sensor_input", 1);
v[5] = array("sensor_input", 2);
v[6] = sum(v[1], v[2], v[3], v[4], v[5]);
v[7] = v[6] / v[0];
```

The outputs are now in v[3] and v[7].

6.2 Execution

In the simplest case, when LibFeature is computing just a single feature vector, the execution model is quite simple: it executes the commands in the bytecode on a temporary array (v[] in the example above), and then copies the lines from the array corresponding to actual features into the feature vector at the end. LibFeature is much more efficient, however, when it is given the opportunity to compute multiple feature vectors in parallel. Every command operates on many feature vectors simultaneously. This vectorization allows the overhead of interpreting the program to be minimized, and LibFeature can start approaching the speed of C code.

For example, suppose that the first few numbers in our sensor_input data happen to be the digits of π: $[3, 1, 4, 1, 5, 9]$. In executing the commands to generate the first four feature vectors, LibFeature would fill in a matrix with the following values:

	Feature Vectors			
Command	1	2	3	4
v[0] = 5;	5	5	5	5
v[1] = array(-2);	NaN	NaN	3	1
v[2] = array(-1);	NaN	3	1	4
v[3] = array(0);	3	1	4	1
v[4] = array(1);	1	4	1	5
v[5] = array(2);	4	1	5	9
v[6] = sum(v[1], ..., v[5]);	NaN	NaN	14	20
v[7] = v[6] / v[0];	NaN	NaN	2.8	4.0

Note that since LibFeature cannot index negative elements in the input array, it replaces these values with NaN. After all of the computations are finished, LibFeature still needs to copy the rows corresponding to output features to the completed feature vectors in memory. When multiple feature vectors are computed at once, individual features of the same feature vector are never contiguous, so the extra copy is always required. While this does take some extra time, it is negligible compared to the time that is saved by being able to compute each of the rows of the temporary matrix all at once, making use of the fact that the elements of each row are consecutive in memory. For example, the inner loop of the code which computes the division command in the last step is actually something like this:

```
for(i=0; i<len; i++) {
   *out = *in1 / *in2;
   out++;
   in1++;
   in2++;
}
```

In practice, as many as 64 or 128 feature vectors are usually computed at once, and modern C compilers are able to make this loop extremely efficient. By vectorizing all of these computations (always doing one step of many feature vectors at once), we

can minimize the overhead of LibFeature. Furthermore, LibFeature takes advantage of vector instructions such as SSE on x86 chips, or AltiVec on PowerPC chips, to make these inner loops run even faster. Using these instructions for many common arithmetic operations speeds up LibFeature by 25% overall in typical usage.

6.3 Multithreading

Since many scientific workstations have dual processors (and other modern CPUs simulate multiple processors using hyperthreading), LibFeature is designed to transparently take advantage of these processors by using two threads for computation.

In cases where all of the input data is to be loaded, using two threads is trivial: one thread loads the first half of the data, and the other thread loads the second half. However, LibFeature allows for validation of individual feature vectors (using the `require` statement in FDL), which complicates matters significantly. Since it is unknown exactly how many feature vectors will be generated, the two threads must communicate frequently in order to write their feature vectors to the same array without leaving any gaps (which would require costly memory moving later, negating the benefit of both threads). Pipes are used both for thread synchronization and communication. The speedup when using two threads on a dual-processor machine is typically 1.5x, though it can be as high as 1.9x for some computationally intensive programs.

7 Benchmarks

LibFeature has been designed to provide no performance penalty in spite of the fact that it interprets its commands. The following benchmarks demonstrate how closely this goal has been achieved.

7.1 Image features

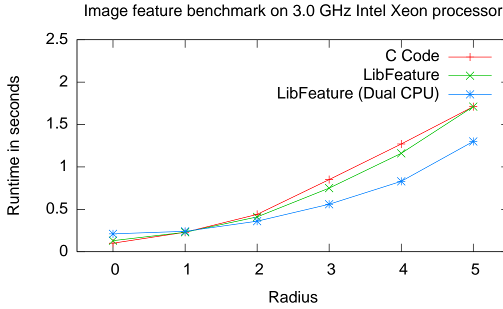
Consider a simple problem where we wish to extract two features from a grayscale 1600×1200 PGM image: the intensity of each pixel, and the statistical variance of an $n \times n$ neighborhood of each pixel. To keep the neighborhood symmetrical, suppose $n = 2{\cdot}radius{+}1$ for some $radius \geq 0$. Here is an example of straightforward C code to implement this feature extraction:

```
void get_features(FILE *fp,
                  float *features, /* output */
                  int radius,
                  int width, int height)
{
   uint8 *input = (uint8 *)malloc(width * height);
   float *p = features;
   int i, j, k;
   int len = (1+(radius*2))*(1+(radius*2));

   fread(input, 1, width * height, fp);

   for(i=0; i<width * height; i++) {
      float sum = 0;
      float sumsq = 0;
      float var;

      for(j=-radius; j<=radius; j++)
         for(k=-radius; k<=radius; k++) {
            int index = i + j + (k*width);
            if (index >= 0 &&
                index < width * height) {
               float v = (float)input[index];
               sum += v;
               sumsq += v * v;
            }
         }

      var = (sumsq - ((sum * sum) / len)) / len;

      *p++ = (float)input[i];
      *p++ = var;
   }
}
```

Now here is the same code, implemented in FDL:

```
(set width (+ 1 (* 2 radius)))
(set len (* width width))
(for i (seq (- 0 radius) radius)
  (for j (seq (- 0 radius) radius)
    (set v (image i j))
    (append array1 v)
    (append array2 (* v v))))
(set sum1 (+ array1))
(set sum2 (+ array2))
(set center (image))
(set var1 (- sum2 (/ (* sum1 sum1) len)))
(set var (/ var1 len))
(output center)
(output var)
```

It requires at most five lines of C code to extract feature vectors from an image given the FDL program above. The LibFeature solution is more compact than the C code, and significantly more



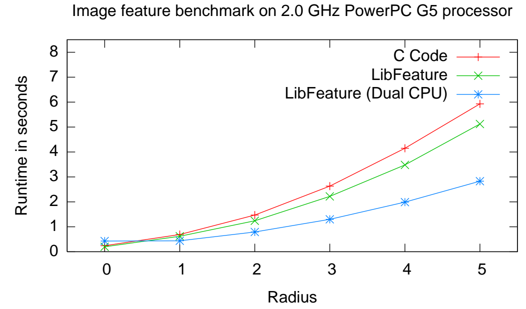


Fig. 1. Benchmark results show that for this particular problem, LibFeature is comparable in speed to straightforward C code when a single processor is used, and somewhat faster when two processors are used.

flexible. To show that there is also no performance penalty, we ran the C code against LibFeature on two different scientific workstations, for several different radii. The following times are measured in seconds:

	Xeon/3.0 GHz		G5/2.0 GHz	
Radius	C	LibFeature	C	LibFeature
0	0.10	0.13	0.24	0.20
1	0.23	0.23	0.69	0.62
2	0.44	0.41	1.47	1.24
3	0.85	0.75	2.63	2.22
4	1.27	1.16	4.15	3.48
5	1.71	1.71	5.93	5.12

Many modern scientific workstations have dual processors. While it is uncommon for a programmer to write multithreaded code for something as simple as feature extraction, LibFeature can take advantage of multiple processors safely and without any extra work on the part of the programmer. While this can introduce extra overhead for small easy problems, in most cases this allows LibFeature to run significantly faster than ordinary C code:

	Dual Xeon/3.0 GHz		Dual G5/2.0 GHz	
Radius	C	LibFeature	C	LibFeature
0	0.10	0.21	0.24	0.43
1	0.23	0.24	0.69	0.44
2	0.44	0.36	1.47	0.79
3	0.85	0.56	2.63	1.30
4	1.27	0.83	4.15	1.99
5	1.71	1.30	5.93	2.83

7.2 FFT

While FDL is not a complete programming language (it does not have any flow control), it is nevertheless powerful enough to compute many complicated algorithms. As an example of this, consider the Fast Fourier Transform (FFT) [5], commonly used to extract frequency features from time-series data. Suppose that you have a million time points, and for every point you wish to create a feature vector containing the Fourier Transform of the n points centered at that point. For simplicity assume that n is a power of two and that we are computing the complex FFT. We implemented a general power-of-two complex FFT in only 46 lines of FDL code, using a very straightforward nonrecursive algorithm. We benchmarked it against FFTW 3.0.1 [6] [7], widely regarded as the one of the fastest FFT implementations available for any computing platform. We tested them both in single-precision mode without making use of multiple processors; the times below are in microseconds, per FFT, when computing at least 100,000 FFTs in a row.

	Xeon/3.0 GHz		G5/2.0 GHz	
FFT Size	FFTW	LibFeature	FFTW	LibFeature
16	0.54	0.94	0.55	0.66
32	1.19	2.00	1.00	1.46
64	2.11	4.63	1.85	3.43
128	4.71	12.15	4.54	8.43

7.3 Analysis of benchmark results

While it is not necessarily unexpected that LibFeature can be more efficient than straightforward C code when using two processors, this does not explain how it matches and sometimes beats the performance of C code even when using a single processor, as seen above. Contributing to LibFeature's efficiency:

1. Cache efficiency: The C code in section 7.1 skips through memory in order to retrieve the values for each feature vector, resulting in inefficient use of the processor's L1 cache. LibFeature typically copies 64 contiguous values at a time from the input array, directly to a row in the temporary matrix.
2. Vector instructions: On both x86 and PowerPC processors, LibFeature uses vectorized addition, multiplication, and division, which allows it to operate on four values at once.
3. Memory mapping: Instead of loading the file into memory in a single `fread` command (which can take some time to complete), LibFeature memory-maps the file, which allows the computation to begin on the first few bytes of the file while the rest is still being loaded.

It is not surprising that FFTW is faster than LibFeature; it has been extensively tuned and undergone several major revisions and has essentially no overhead. The fact that LibFeature is only a factor of 2-3 slower than FFTW, given only 46 lines of high-level FDL code, is impressive and serves as validation of LibFeature's design and architecture. Note of course that LibFeature has more overhead and must be given thousands of FFTs to compute in order to approach this speed. Also, to be fair, LibFeature consumes vastly more memory than FFTW needs to for the same task.

8 Conclusions and Future Work

LibFeature is designed to make life easier for the machine learning or data mining researcher. Instead of wasting time writing complicated transformations to construct feature vectors from input data, you can use LibFeature to do this work for you. By abstracting the feature vector generation from the algorithm, it becomes easier to change the feature vectors on the fly; experimenting with new ideas for features thus requires less effort. Because LibFeature was implemented very carefully with performance in mind, it is usually possible to use LibFeature and pay no performance penalty at all. In fact, in many circumstances, LibFeature is significantly faster than straightforward alternatives.

We are making use of LibFeature in several projects, and many new capabilities are constantly being added. Some possible new capabilities that we are considering for the future include better support for reading numerical data from ASCII text files, more binary data formats, more language bindings, dual-pass algorithms for normalized features, integer features, and efficient random shuffling of vectors.

9 Acknowledgements

Thanks to Dennis DeCoste for the initial idea and supporting this work, and to Ben Bornstein for helpful suggestions on this paper. The research described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

10 Software Availability

Information on obtaining LibFeature is available from the JPL Machine Learning Group web page: `http://ml.jpl.nasa.gov/`

References

1. Date, C.J., Darwen., H.: A Guide to the SQL Standard, 4th Edition. Addison-Wesley (1997)
2. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison Wesley (1997)
3. Gosling, J.: Java intermediate bytecode. In: Proceedings of the Workshop on Intermediate Representations ACM SIGPLAN Notices. Volume 30. (1995) 111 – 118
4. Post, R.J.: OpenGL Shading Language. Addison-Wesley (2004)
5. Cooley, J., Tukey, J.: An algorithm for the machine computation of the complex fourier series. Mathematics of Computation **19** (1965) 297–301
6. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. Volume 3., Seattle, WA (1998) 1381–1384
7. Frigo, M., Johnson, S.G.: FFTW 3.0.1 (software) (2003) `http://www.fftw.org/`.