ivl by Example

A Scientific Computing Library, Applied to Learning and Vision

Kimon Kontosis, Yannis Avrithis and Christos Varytimidis
National Technical University of Athens
9, Iroon Polytechniou Str., 157 73 Zographou, Athens, Greece
{kimonas,iavr,chrisvar}@image.ntua.gr

ABSTRACT

Researchers, engineers and professionals in any discipline of scientific computing often prefer a computing and visualization environment like MATLAB[1] for everyday work, especially when trying out a new idea. On the other hand, joint development of large projects with emphasis on performance, ease of integration, or genericity, typically requires a full-fledged language like C++, particularly so for open source, cross-platform software. Transition between these two worlds often ends up in painful code re-writing.

Our approach is to bring, among others, elements of the MATLAB syntax into C++, where they can become even more powerful, blended *e.g.* with classes, templates, or the standard library (STL). Our open source template library `ivl` extends language syntax towards mathematical notation and serves as a foundation in scientific computing rather than a collection of problem-oriented algorithm implementations. As such, we choose to illustrate it through a concrete example in the field of machine learning and computer vision, followed by a brief outline of its design principles and major features.

1. ORIENTATION

Instead of explaining *what* `ivl` *is*, we shall attempt in section 2 to show *what it looks like* by building a *randomized decision forest classifier* from scratch. Trained by one input image and associated ground truth of pixel-level class labels, the classifier learns to classify individual pixels of a new image into one of 23 object classes like 'building', 'flower', or 'car', plus a 'void' (unknown) class, according to the MSRC dataset[2]. In particular, we follow the formulation of Shotton *et al.* [3] and Moosman *et al.* [2].

We assume familiarity with MATLAB and a good knowledge of C++. On the other hand, no background on the classification method itself is assumed; we shall explain it along the way, using blocks of actual working C++ code instead of algorithm pseudocode and, in a few cases, mathematical formulae. In less than six pages, we shall expose the complete 120-line project code, including its `main` function, while discussing relevant `ivl` features.

In fact, using a set of `bash`/`awk`/`sed` scripts, the single project source file is automatically generated, compiled and executed from the same file as the `latex` source of this document. The documentation of `ivl` follows the same principle, validating each piece of code with the compiler, as well as feeding the standard output of the executable back to the document for each individual example.

[1] http://www.mathworks.com/products/matlab/
[2] http://research.microsoft.com/en-us/projects/objectclassrecognition/

2. A RANDOMIZED DECISION FOREST

We begin in section 2.1 with preliminaries like including headers and defining types to represent data. We then discuss in sections 2.2 and 2.3 the classification and learning processes, respectively. Although we are certainly not building a generic library here, our development is such that generalizing to arbitrary learning problems is straightforward. Specific aspects of the particular image classification problem are handled in section 2.4, and the `main` function is given in section 2.5.

2.1 Preliminaries

2.1.1 *Using* `ivl`

We first include a couple of `ivl` headers, specifying that we shall also be using OPENCV[3] to load or display images,

```
#include <ivl/ivl>
#include <ivl/cv>
```

To keep code structure as simple as possible, we define a set global variables, classes and functions within a `forests` namespace, which uses namespace `ivl`,

```
namespace forests {
using namespace ivl;  using ivl::rand;
```

2.1.2 *Patches and samples*

Each image pixel is classified according to its surrounding square *image patch p*. A patch thus plays the role of a *data point* in our problem, and is defined as a contiguous rectangular *sub-array* of an image,

```
typedef image<double>::const_box patch;
```

where `image` is an `ivl` template class that is interoperable with OPENCV's `IplImage`; `double` here is the underlying data type of each color component of each pixel, and `box` is

[3] http://opencv.willowgarage.com/

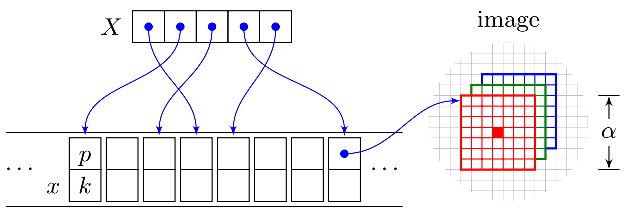

Figure 1: Memory organization for patches, samples, and data sets.

a *view* that can be used to access the actual image data as if it were an image itself—respectively, `const_box` is a read-only view. The patch side α in pixels and patch size $3\alpha^2$ for three color channels are fixed here,

```
const int alpha = 9;              // patch side
const int sz = 3 * _[alpha]->*2;  // patch size
```

where syntax `_[]` denotes an `ivl` `scalar`, providing, among others, *power* operator `->*` to fundamental types like `int`.

A *sample* $x = (p, k)$ is a pair of a patch p and a discrete *target variable* or *class label* k,

```
struct sample { patch p;  int k; };
```

while a set of samples or *data set* X is represented by a *sub-array* of a given array of samples,

```
typedef array<sample>::const_indirect_array data;
```

where `array` is `ivl`'s basic one-dimensional array type; `sample` here is the underlying data type, and `indirect_array` is another type of view that is non-contiguous in memory, indexed by an arbitrary array of positions—again, `const_indirect_array` is read-only. Data in all array types can be accessed in a variety of ways, including different types of iterators that are either compatible or generalize those of the STL.

The overall memory organization is illustrated in Figure 1. A data set X can be seen as an array of pointers to samples $x = (p, k)$, where the samples are actually stored in another array and each patch p can also be seen as a pointer to a rectangular 3d box of a 3-channel image. When sampling an image, a patch is constructed with reference to the central pixel; for that, α is an odd integer. Nothing fancy so far, but we shall see how conveniently our structures are initialized and used.

2.1.3 *Tests*

Randomized decision forests [1] consist of a number of *decision trees*, each trained separately on a number of random tests. A *test* t is an object that comprises one or two simple *attributes* or *features* ϕ_1, ϕ_2 of an image patch, a *split function* f, and a threshold θ. A feature refers to a specific color component of one pixel of the patch, represented by a position in the underlying sub-array.

```
struct test {
   int type, feat[2];  double th;
```

There are four *types* of tests depending on the choice of f, as specified in Table 1. Being as straightforward as possible, we define f as follows.

```
   double f(const sample& x) const {
      double v1 = x.p[feat[0]], v2 = x.p[feat[1]];
      return type == 0 ? v1 : type == 1 ? v1 + v2 :
             type == 2 ? v1 - v2 : abs(v1 - v2);
   }
```

type	value	sum	difference	absolute
$f(x)$	v_1	$v_1 + v_2$	$v_1 - v_2$	$\lvert v_1 - v_2 \rvert$
example	▦	▦ + ▦	▦ − ▦	| ▦ − ▦ |

Table 1: Let v_1, v_2 be the values of two features on the patch of a given sample x. The value of $f(x)$ for each of the four types of tests is shown here, along with an example on tiny patches.

A random test is generated by choosing all parameters uniformly at random within appropriate ranges,

```
   void gen(const data& X) {
      type = rand(0, 3);  lnk(feat) = rand(2, 0, sz-1);
      th = rand[extrema( _[this][&test::f](X) )];
   }
};
```

where our first exotic `ivl` examples come up. Function `rand` yields either a scalar or a 2-element `array` that is *linked* to C array `feat`. Given a data set X, the range of thresholds θ is determined by the extrema of $f(x)$ over all samples $x \in X$. Here `&test::f` is a C++ *pointer to member* function `f` of class `test`, and syntax `_[this][&test::f]` denotes an `ivl` *element function* that can apply f to the entire data set X—what one would write as $f(X)$ in set theory!

But `extrema` yields two quantities (min, max)—how are they both fed into `rand`? Let us wait until we build our own example in sections 2.3.1,2.3.2.

2.1.4 *Class distributions*

Fixing the number K of classes,

```
const int K = 24;  // number of classes
```

we represent a distribution of *class probabilities* $P(k)$ for $k = 1, \ldots, K$ by an `ivl` `array` of *fixed length* K,

```
typedef array<double, fixed<K> > prob;
```

which is allocated on *stack*, contrary to the default `ivl` `array` that is allocated on *heap*. Despite implying more copies and no resizing, this option is generally preferred for small arrays, saving allocation time. In general, the second template argument of class `array` can used to specify several more options on representing and manipulating underlying data.

2.2 Classification

We first discuss the decision tree classification process in section 2.2.1, and then generalize to decision forests in section 2.2.2.

2.2.1 *Decision trees*

A single *decision tree* classifier is a binary tree consisting of *split nodes* and *leaf nodes*. Both inherit a *base node*, capturing only common behavior,

```
struct node
   { virtual const prob& classify(const sample&) = 0; };
```

allowing us to define a *tree* as a pointer to its *root* node,

```
typedef node* tree;
```

A *split node*, apart from pointers to its left and right *children* l, r, is assigned a test t, in turn equipped with split function f and threshold θ,

```
struct split_node : public node {
   test t;  node *l, *r;  // left / right children
   split_node(test t,node *l,node *r): t(t),l(l),r(r) {}
```

Starting from the root node, a new sample x recursively descends the tree towards l (r) whenever $f(x) < \theta$ ($f(x) \geq \theta$), as illustrated in Figure 2,

```
   const prob& classify(const sample& x)
      { return (t.f(x) < t.th ? l : r)->classify(x); }
};
```

eventually reaching a *leaf node*, for instance node 5 in Figure 2. Each leaf node n is assigned an array of learned *class probabilities* $P(k|n)$ for $k = 1, \dots, K$,

```
struct leaf_node : public node {
   prob P;  leaf_node(const prob& P) : P(P) {}
```

which is returned by the final call to `classify`,

```
   const prob& classify(const sample&) { return P; }
};
```

If we were to classify sample x based on a single tree, we would simply let x descend the tree down to some leaf node n and choose the class k that maximizes $P(k|n)$, for instance class 2 in Figure 2.

2.2.2 *Decision forests*

A single tree classifier is prone to *overfitting* due to hard decision boundaries. To improve *generalization*, a *decision forest* classifier F uses an array of T trees,

```
typedef array<tree> forest;
const int T = 4;  // number of trees
```

Now a sample x descends *all* T trees down to leaf nodes $N = \{n_1, \dots, n_T\}$, and is classified to the most likely class

$$k^* = \arg\max_k P(k|N) = \arg\max_k \sum_{n \in N} P(k|n) \qquad (1)$$

after averaging (or summing) class distributions over leaf nodes,

```
int classify_(const forest& F, const sample& x)
   { return arg_max(sum( F[&node::classify](x) )); }
```

where we apply, to sample x, member function `classify` of class `node` for forest F, an array of (pointers to) nodes, yielding an array of entire leaf class distributions! Function `sum` then operates along this array, computing a single distribution, whose mode is found by `arg_max`.

We can now `classify` an entire data set X with a single call to a *named* element function

```
static binary_elem<int, const forest&, const sample&,
   classify_> classify;
```

to be used in function `main` on all pixels of an image. This syntax denotes a binary function object named `classify` with return type `int`, two input arguments of types `const forest&` and `const sample&`, and underlying implementation given by function `classify_`. An operator `()` provides the input arguments, which may be arrays as well. For instance, given a data set X, `classify_` will be invoked once for every sample $x \in X$.

2.3 Learning

The decision tree learning process is illustrated in Figure 3 for a toy 2d classification problem of $K = 5$ classes, where data points are shown as points $x = (x^1, x^2) \in \mathbb{R}^2$, colored according to class. Learning relies on *data splitting*, which for this problem is represented by lines separating the plane into two half-planes.

Starting from the entire data set, five random tests are generated with split functions of the linear form $f(x) = f(x^1, x^2) = ax^1 + bx^2$. The one that best separates the data into classes is chosen to split the data set into two subsets that are recursively subjected to the same process until no split is considered necessary. A split node in the tree records each chosen test, while leaf nodes record the class label distributions of the data points they contain. The resulting tree is not necessarily balanced, and training samples may still be misclassified.

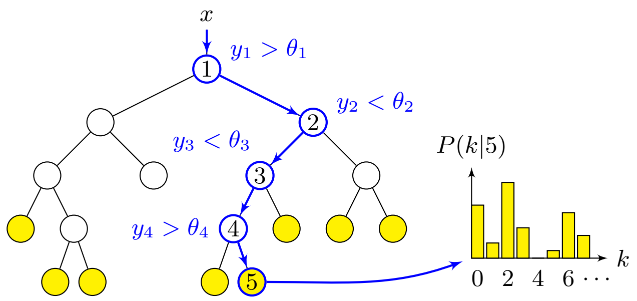

Figure 2: Decision tree classification. Sample x descends from root node 1 down to leaf node 5, assigned class probabilities $P(k|5), k = 1, \dots, K$. Each node $i = 1, \dots, 5$ is assigned a test with split function f_i and threshold θ_i, and we let $y_i = f_i(x)$. Leaf nodes are shown in yellow.

We first discuss data splitting in section 2.3.1, and then develop randomized decision tree and forest learning in sections 2.3.2 and 2.3.3, respectively. Once the form of split functions f is fixed, the remaining process is independent of the dimensionality or structure of the input space, hence applicable in particular to the 2d problem of Figure 3, to image classification or indeed to any classification problem. Learning is a lot more complex than classification, providing a good opportunity to reveal a few more advanced features of `ivl`.

2.3.1 *Data splitting*

Given a data set X and a test t with split function f and threshold θ, we compare $f(x)$ to θ for each sample $x \in X$, and *split* (partition) X into L, R, where $L = \{x \in X : f(x) < \theta\}$ and $R = X \setminus L$,

```
ret<data, data> split_(const test& t, const data& X) {
   array<bool> mask = _[t][&test::f](X) < t.th;
   return (_, X[find(mask)], X[find(!mask)]);
}
```

where we apply function f of test t to the entire set X, as we did in `test::gen`; then, collect all test comparisons in a boolean array, or *mask*; use `find` as in MATLAB to index X and split it into L, R; and return both as *output arguments*! This is made possible by construct `(_,L,R)`, an `ivl tuple` representing pair (L, R); and `ivl` *return type* `ret`, which, more than being a tuple itself, *eliminates temporary objects*. We are now ready to `split` X according to an entire *array* of tests **t** by

```
static binary_elem<ret<data, data>, const test&, const
   data&, split_> split;
```

Trying a number of *random splits*, the strategy is to find the one that best separates classes across L, R. For this we first need class label distributions, empirically estimated as *normalized histograms* over an array of labels,

```
prob hist(const array<int>& k)
```

```
   { prob h(K, 0.0);  ++h[k];  return h / k.size(); }
```

which is exactly how MATLAB syntax becomes richer within C++, observing how much is hidden beneath the 6-character expression `++h[k]`: whenever a label is repeated in array `k`, the *reference* to a bin counter in `h[k]` is repeated as well, and `++` just increases it!

Next, given a data set X with class label distribution $P(k) = P(k|X)$ empirically estimated by a normalized histogram, its *entropy*

$$H[X] = -\sum_{k=1}^{K} P(k) \ln P(k) \tag{2}$$

is found as simply as

```
double entropy(const prob& P)
   { return -sum( (P * log(P))[P>0] ); }
```

Most STL math functions like `log`, and *all* arithmetic, comparison, logical, bitwise and compound assignment C++ operators like (element-wise) `*`, are extended for *all* `array` types in `ivl`. What's more, every `ivl` function or operator actually constructs an element function that *defers evaluation* until needed. As a result, *no temporaries* are constructed for the entire expression: what is really happening behind the scenes is that the analogous expression of individual elements $P(k)$ is constructed at compile time and a *single* `for` loop over k is executed at run time!

Condition `P>0` simply encapsulates the fact that $x \ln x \to 0$ as $x \to 0+$, so that zero probabilities do not contribute to the sum, which is an implicit assumption in definition (2). Operator `[]` applies equally to `ivl` arrays, function objects, or entire expressions, simply because all are arrays, differing only on template parameters!

Hence, given a data set X (or L, R in particular), the entropy of its class label distribution is given by

```
double H(const data& X)
   { return entropy(hist( X[&sample::k] )); }
```

But how do we get class labels out of X? For all that matters, X is an array of samples, so we *view* member `k` of class `sample` for each sample $x \in X$ as if they all formed an array of labels!

Now, given a split $\mathcal{Y} = \{L, R\}$ of data set X, its *information gain* over X is given by *mutual information* $I[X; \mathcal{Y}] = H[X] - H[X|\mathcal{Y}]$, where

$$\begin{aligned} H[X|\mathcal{Y}] &= -\sum_{Y\in\mathcal{Y}} \sum_{k=1}^{K} P(k,Y) \ln P(k|Y) && (3) \\ &= \sum_{Y\in\mathcal{Y}} P(Y) \left\{ -\sum_{k=1}^{K} P(k|Y) \ln P(k|Y) \right\} && (4) \\ &= \sum_{Y=L,R} \frac{|Y|}{|X|} H[Y] && (5) \end{aligned}$$

is the *conditional entropy* of X given $\mathcal{Y}$ and $|X| = |L| + |R|$. Since $H[X]$ is independent of the split, the gain is maximized whenever $-H[X|\mathcal{Y}]$ is,

```
double gain_(const data& L, const data& R) {
   int l = L.size(), r = R.size();
   return !(l&&r) ? -infty : -(l*H(L) + r*H(R)) / (l+r);
}
```

where a value of $-\infty$ signifies that the *trivial split* $\mathcal{Y} = \{\emptyset, X\}$ is *not* an option. In fact, $H[X|\mathcal{Y}]$ is the expected

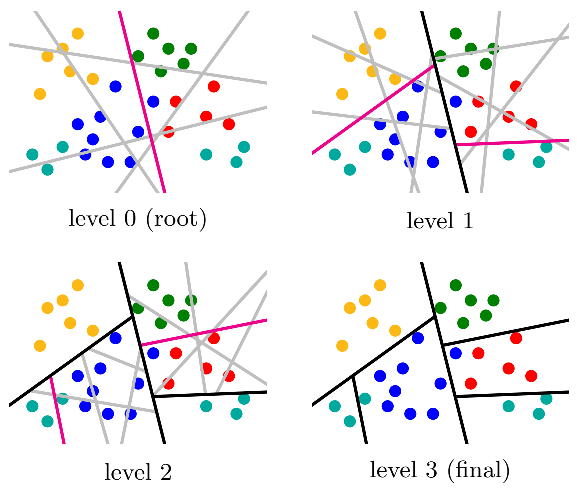

Figure 3: Decision tree learning for a 2d problem at each tree level. (Gray) random splits (lines); (magenta) best splits found for each node at current level; (black) best splits of previous levels, and final ones.

(average) entropy of X when split according to $\mathcal{Y}$, so we do not split X unless $-H[X|\mathcal{Y}]$ is below *gain threshold* γ,

```
const double gamma = -0.1;  // gain threshold
```

We certainly want to be able to compute the gain of multiple splits at once, given by two *arrays* of data sets L, R of the same length,

```
static binary_elem<double, const data&, const data&,
   gain_> gain;
```

which is yet another convenience of an element function.

2.3.2 *Tree learning*

We are now in a position to define the entire tree learning process, as illustrated in Figure 3. Given a data set X, we shall try a number of random tests, each giving rise to a split, looking for the one that best separates classes according to our measure of information gain. Fixing the number S of tests, or splits,

```
const int S = 10;  // number of random tests / splits
```

we begin by generating an array $\mathbf{t}$ of random tests,

```
node* learn(const data& X) {
   array<test> tests(S);  tests[&test::gen](_[X]);
```

where we apply member function `gen` of class `test` to *array* $\mathbf{t}$ of tests, giving an *array* X of samples as input! But X is wrapped inside a `scalar` by `_[]`, thus treated as a single object rather than an array of objects, such that the loop over t in $\mathbf{t}$ is the outer one.

Next, we split X according to each test t in $\mathbf{t}$, measure its gain, and maximize over t,

```
   double g;  int best;  // max gain, best test
   (_, g, best) = max++( gain[split(tests, _[X])] );
```

where element functions are connected into a processing *pipeline* to do all the work in one line of code! Again, there are no temporaries for the entire expression: there is only one `for` loop over t underneath, and for each t, an inner loop over samples $x \in X$. And again, the loop over t becomes the outer one by wrapping X inside a scalar.

Observe that `split` returns a `tuple`, while `gain` expects two separate arguments L, R; syntax `gain[]` conveniently provides a copy-free conversion. This is exactly the kind of 'collaboration' that `extrema` and `rand` had within `test::gen` in section 2.1.3. By default, every `ivl` function has a fixed output type—in particular, the maximum value of an array for `max`. An extended `++` version enables *output argument overloading*! For instance, another option is a tuple containing the maximizing index as well.

All that remains now is to keep splitting recursively and recording our actions in split nodes, until the class label distribution is pure enough or no more gain is possible (*e.g.*, we are left with one sample),

```
   if(g < gamma && !isinf(g)) {
      data L, R;  const test& t = tests[best];
      (_, L, R) = split(t, _[X]);
      return new split_node(t, learn(L), learn(R));
   }
```

in which case we just record the residual class label distribution in a leaf node,

```
   return new leaf_node(hist( X[&sample::k] ));
}
```

2.3.3 *Forest learning*

The learning process described so far refers to a single tree; the element function that will lead to *forest* learning,

```
static unary_elem<tree, const data&, learn, loops::
   intensive> core_learn;
```

has the optional property `loops::intensive`: given an *array* of T trees, `core_learn` automatically distributes the work load over *multiple processing cores* whenever possible, instead of simply iterating over trees in a single core! In fact, namespace `ivl::loops` is the place where all `for` loops are isolated, potentially executing concurrently while leaving all remaining code in and out of `ivl` as clean as possible.

Then, starting with a single data set X, all we need to do is artificially *replicate* it into an array of T *virtual copies* of X, and feed this array into `core_learn`,

```
forest learn(const array<sample>& X)
   { return core_learn( rep(T, data(X)) ); }
```

where the constructor of `data` is needed to generate a *view* of the given array of samples. Ideally, each CPU core will learn a different randomized decision tree! Given multiple training images, one would employ different, possibly overlapping, random subsets of the data for each tree; but we keep things as simple as possible and train the classifier on a *single* image.

2.4 Image classification

We now complete the remaining operations required for the collection of samples from pairs of *input images* and *class label maps* for training, and the construction of *label images* from classifier output. The *ground truth* consists of a manually generated image of pixel class labels for each input image, either used for training, or for evaluating classification performance. These images are color-coded, so we shall provide *class label map* ↔ *label image* conversions.

It is convenient to group an input image and a class label map into a structure named, well, quite unimaginatively, *group*,

```
struct group { image<double> im;  array_2d<int> id; };
```

id	class	rgb	color	id	class	rgb	color
0	void	000	⬛	12	car	102	🟪
1	building	200	🟥	13	bicycle	302	🟪
2	grass	020	🟩	14	flower	122	🟦
3	tree	220	🟨	15	sign	322	🟥
4	cow	002	🟦	16	bird	010	🟩
5	horse	202	🟪	17	book	210	🟫
6	sheep	022	🟦	18	chair	030	🟩
7	sky	222	⬜	19	road	212	🟪
8	mountain	100	🟫	20	cat	032	🟩
9	aeroplane	300	🟥	21	dog	232	🟩
10	water	120	🟩	22	body	110	🟫
11	face	320	🟧	23	boat	310	🟧

Table 2: MSRC class label ids, names, rgb values (divided by 64), and colors.

where `ivl array_2d` is an optimized array of fixed dimensionality 2. By contrast, the dimension of a general `ivl array_nd` is a property rather than a static template parameter, and may change after construction. In fact, `ivl image` is an `array_nd` with an arbitrary number of color channels in its third dimension. Every `ivl` array type can also be seen as an one-dimensional `array`, can access the underlying data as stored in memory and can be *reshaped* without actual data reallocation.

2.4.1 *Sampling*

We represent an image *position* by an `ivl` 2d `point`, assigned a row `r` and a column `c`, also referred to as coordinates `y`,`x`. Given a group g and position q, we find the range of image rows and columns for a square patch of side α pixels that is centered at q,

```
sample pick(const point<int>& q, const group& g) {
   sample x;  int h = alpha / 2;  // half-side
   x.p = g.im((q.r-h,_,q.r+h), (q.c-h,_,q.c+h), _);
```

where `(a,_,b)` denotes an `ivl range` equivalent to MATLAB's `(a:b)`. We then use the ranges to refer indirectly to all three color channels of this patch in the input image, with `ivl` syntax `a(i,j,_)` providing *multidimensional array indexing* equivalent to MATLAB's `a(i,j,:)`; however, the resulting sub-array is a *view* rather than a copy, since a patch is defined as such.

Of course, indexing works as well for single coordinates or just a point. This is the case for the class label image, which we only sample when training,

```
   if(!g.id.empty()) x.k = g.id(q);
   return x;
}
```

During learning, an image is *sparsely sampled* on a grid of overlapping patches, with a grid step σ fixed here,

```
const int sigma = 5;  // grid step
```

Given a group g, we specify the rows and columns of grid positions, leaving a margin of half patch side around the images of g,

```
array_2d<sample> sparse(const group& g, int s = sigma) {
   int h = alpha / 2;  // half-side
   size_range rows = (h,_,s,_,g.im.rows()-h-1);
   size_range cols = (h,_,s,_,g.im.cols()-h-1);
```

with `(a,_,b,_,c)` being equivalent to MATLAB's `(a:b:c)` (a range from `a` to `c` with step `b`). We then sample g at all positions on the grid by

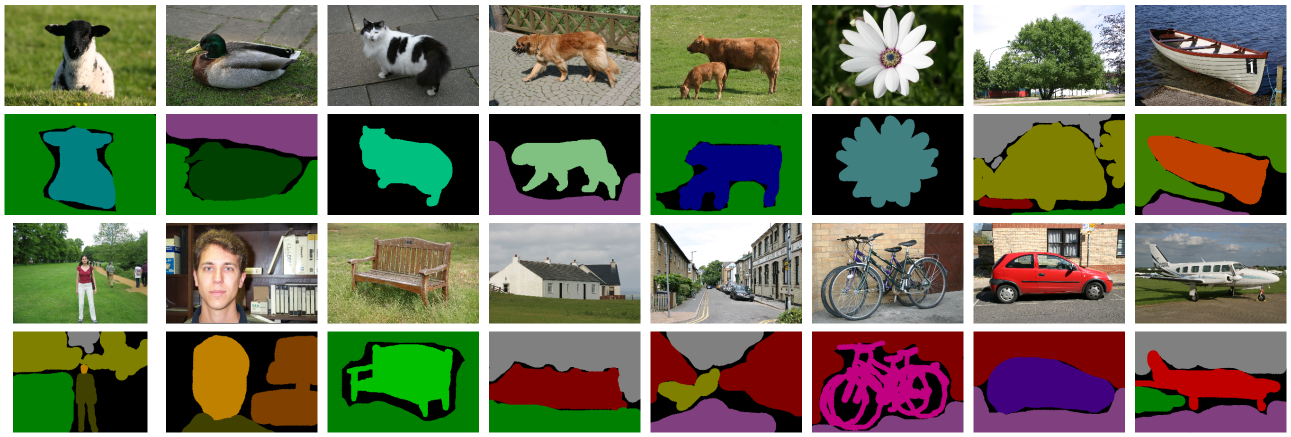
Figure 4: Example images from the MSRC dataset. Rows 1 and 3 rows depict input images, while rows 2 and 4 depict the associated ground truth, representing class labels by the color codes of Table 2.

```
   return _[pick]( pnt[grid(rows, cols)], _[g] );
}
```

In doing so, we construct an ivl 2d grid comprising two 2d coordinate arrays, which we couple into a single 2d array of points by pnt; grid is similar to MATLAB's meshgrid but with row-first ordering, as in matrices. Wrapped inside _[], pick becomes an element function that now applies to a 2d array of points and, maintaining their size and shape, yields a 2d array of samples! By contrast, g is treated as a scalar as in previous examples. Whether one uses a temporary element function by _[] or defines a named one is really a matter of taste and intended usage.

On the other hand, classification typically involves *dense sampling* of one patch centered on every pixel. Thanks to our definition above, dense sampling is no more than sparse sampling of step 1,

```
array_2d<sample> dense(const group& g)
   { return sparse(g, 1); }
```

2.4.2 *Class label coding*

Class label images are typically *color-coded* images of pixel class labels of the same size as input images. Class labels and color codes for MSRC are shown in Table 2. Observing that rgb codes take values in $\{0, 1, 2, 3\}$ when divided by 64, we can encode each rgb tuple by a single base-4 integer,

```
const int c[K] =
   {  0, 32,  8, 40,  2, 34, 10, 42, 16, 48, 24, 56,
     18, 50, 26, 58,  4, 36, 12, 38, 14, 46, 20, 52  };
const array<int> colors(c);
```

where we conveniently initialize an ivl 1d array by a C array. This is an array of length $K = 24$, mapping class labels to color codes; the inverse mapping is given by

```
array<int> labels() {
   array<int> id(max(colors) + 1, 0);
   id[colors] = (0,_,K-1);  return id;
}
```

making use of indirect array indexing and ranges, exactly as in MATLAB. With the two mappings at hand, we can easily provide for *image* $\rightarrow$ *label map* conversion by dividing color channels by 64, converting from base-4 to base-10 and mapping to label,

```
array_2d<int> im2label(const image<unsigned char>& im) {
   return labels() [ im(_,_,0) / 64 +
                        im(_,_,1) / 16 + im(_,_,2) / 4 ];
}
```

where, as in MATLAB, im(_,_,i) is the complete i-th color channel of image im. Similarly, for *label map* $\rightarrow$ *image* conversion, we map to color code, convert back to base-4, and multiply by 64,

```
image<unsigned char> label2im(const array_2d<int>& id) {
   image<unsigned char> im(id.rows(), id.cols(), 3);
   array_2d<int> c = colors[id];
   im(_,_,2) = c >> 4;  im(_,_,0) = c;
   im(_,_,1) = c >> 2;  im &= 3;  return im <<= 6;
}
```

where we now choose to take advantage of *bitwise* operators. Yet, this may be considered kind of 'wordy'. State of the art ivl code would do the same in just one line of code, but is still experimental.

And that's it! We can now close the forests namespace.

```
}  // namespace forests
```

2.5 Main program

What is left for the main program is really simple. We begin by declaring the use of namespaces ivl and forests,

```
int main(int argc, char** argv) {
   using namespace ivl;
   using namespace forests;
```

Then, assuming that the first two program arguments are the filenames of an input image and its ground truth for training, we load them into a group and display them

```
   group g;  g.im = imread<double>(argv[1]);
   g.id = im2label(imread<unsigned char>(argv[2]));
   imshow(g.im, "training image");
   imshow(label2im(g.id), "ground truth");
```

with template versions of MATLAB's or OPENCV's commands imread, imshow, internally using OPENCV. Finally, we train a decision forest on sparse samples of the single input image,

```
   forest F = learn(sparse(g));
```

and classify another input image given as a third argument,

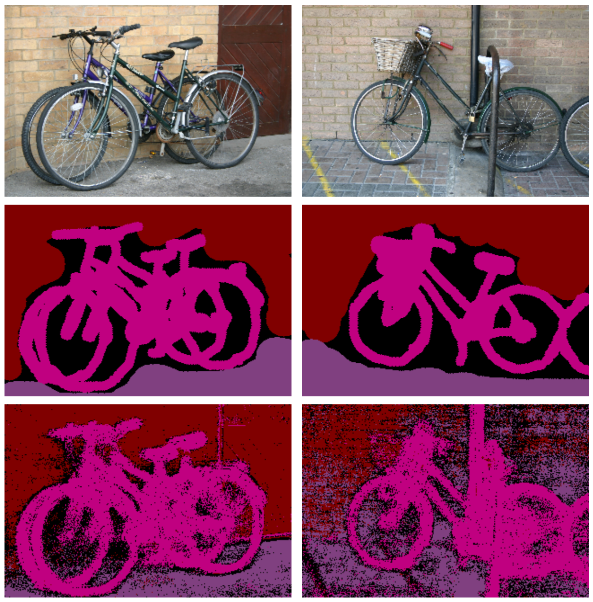

Figure 5: Classification for images 8 (left) and 9 (right) of sequence 8 ('bicycle'), with training on image 8 only. (Top) input images; (middle) ground truth; (bottom) classification labels.

hopefully of similar class labels,

```
    g.im.load(argv[3]);  g.id = _;
    imshow(g.im, "test image");
    g.id = classify(F, dense(g));
    imshow(label2im(g.id), "classification");
}
```

where again, `classify` maintains the size and shape of its input samples, yielding a 2d array of labels! Syntax `a=_` just clears array `a`.

Let us run the program! Figure 4 shows a few samples of the MSRC dataset; we first choose the 'bicycle' sequence. Figure 5 shows the result of two runs, on the same image used for training and a different one. Training and testing take approximately 500 ms and 50 ms, respectively. Another example from the 'road'/'building' sequence is given in Figure 6. Observe that 'sky' is correctly classified even where ground truth incorrectly says otherwise. However, data for 'car' are far too few for the classifier to learn: it fails even in the training image!

This completes our example of a very basic but complete working version of a randomized decision forest classifier for images. The result is pretty good for the simplicity of the method, small size of the training set, and small number of tests, $S = 10$. Extending to multiple training images, other datasets, or performance evaluation, is left to the reader. For really more than a toy, one may consult [3], [2].

3. SO WHAT IS `ivl`?

`ivl` is a fully templated C++ library with convenient syntax for arrays, matrices, images, tuples and functions, including mathematical operations upon them. Often resembling a new language, it targets abstract, concise, readable, yet efficient code. It supports the principle that the path from theory and pseudocode through rapid prototyping to 'production quality' software should be as short as possible.

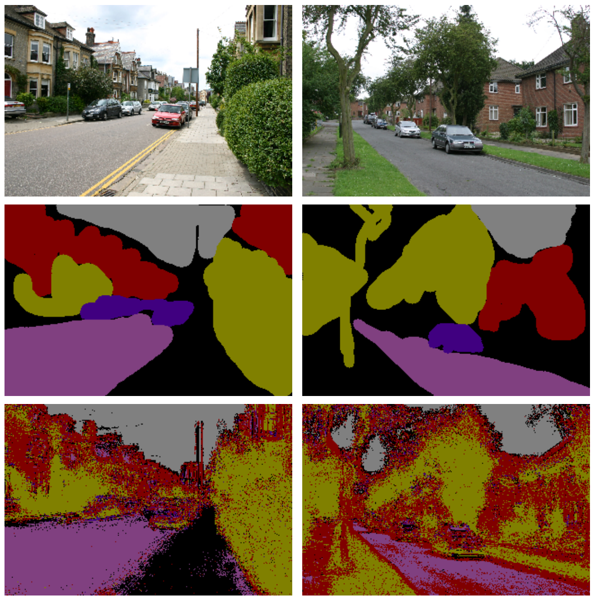

Figure 6: Classification for images 23 (left) and 26 (right) of sequence 17 ('road'/'building'), with training on image 23 only. (Top) input images; (middle) ground truth; (bottom) classification labels.

`ivl` consists of the core library and, currently, two modules: `ivl-lina`, wrapping LAPACK[4] linear algebra functions, and `ivl-cv`, using OPENCV for image processing and computer vision. The `ivl` core library does not depend on any external libraries. It is compilable with most modern ISO C++98 compilers and automatically configured and installed on any operating system using CMAKE. The entire core library has no single piece of non-header code, while pre-processor macros are strictly limited to platform and module options.

The core library uses a number of advanced meta-programming techniques including a modification of *curiously recurring template pattern* (CRTP), and a number of performance tools such as STL-compatible iterators, *n-dimensional iterators*, automatically-controlled *lazy evaluation* and *dynamic multithreading*. Based upon templates, the compiler is utilized to solve meta-problems related to how data can be stored or copied in the most efficient way. Along with inlining and compiler optimization, this results in transforming tedious loops into high-level code that is often equivalent to the optimum C code in performance. No more space is allocated for `ivl` objects than absolutely necessary.

The fundamental class of `ivl` is `array`. It is templated on the type of data it holds, *e.g.* `array<int>`, `array<float>`, and on how it manipulates data, *e.g.* `array<T, fixed<N> >`, `array<T, subarray<A,I> >`. There is a class hierarchy having `array` as the base class, `array_nd` as the n-dimensional derived class, and `array_2d`, `image` as higher level derived classes. Each inheritance layer has added functionality, *e.g.* `array_2d` supports *matrix multiplication*. n-dimensional arrays can be viewed as one-dimensional as defined by their base class `array<T,D>`, with 'unfolded' dimensions.

Operators on the underscore character `_` conveniently construct *e.g.* a `tuple`, a `range` or a function object, while *com-*

[4]http://netlib.org/lapack/

pound operators beyond C++ support are defined including array concatenation, matrix multiplication, left/right division, power, and (conjugate) transpose. Tuples equip function objects with *left/right argument overloading* and *multiple return values* without copies. Function *pipelining* and *serializing* makes code more abstract and extremely compact, eliminating temporary storage and taking care of the underlying, optionally parallel, implementation.

`ivl` is at the moment in a pre-release version where corrections, feature completions and performance improvements are constantly taking place. However, all syntax is fully working and well optimized. Its documentation includes a leisurely introduction by example, a complete reference, and a number of samples, with this document not being an exception.

Apart from integrating with more scientific libraries for *e.g.* numeric computation or visualization, there is also a lot of undergoing development in directions like *tensor notation* and *functional programming*, which can give entirely new perspectives to scientific computing. Massively parallel computation on GPU is not supported yet but may follow with minimal effort, given that *e.g.* `for` loops are very limited and completely isolated.

We hope you have enjoyed this quick tour of `ivl` and welcome you to try it, starting from its project home[5], or at SourceForge[6]. There is much more to explore!

4. REFERENCES

[1] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[2] F. Moosmann, E. Nowak, and F. Jurie. Randomized clustering forests for image classification. *PAMI*, 30:1632–1646, 2008.

[3] J. Shotton, M. Johnson, and R. Cipolla. Semantic texton forests for image categorization and segmentation. In *CVPR*, 2008.

[5] `http://image.ntua.gr/ivl/`

[6] `http://sourceforge.net/projects/ivl/`