Documentation for **imagetrans**

Peter Carbonetto
pcarbo@cs.ubc.ca

April 28, 2003

Introduction

This document presents a brief overview of the **imagetrans** package. I *highly* recommend you read this document before using or modifying the code I wrote. I did my best to document the code fairly well but the fact is I did not cover all the bases – there are some warnings, caveats and advice that did not make its way into the function help and comments.

The first thing you may be wondering is, what in the heck is **imagetrans**? It is an environment for the purpose of building and testing statistical generic object recognition models using semantically- labeled data. If you want to find out more about the theoretic (and not so theoretic) underpinnings of all this code, I refer you to three papers, a couple of which I wrote. These papers are

- Pinar Duygulu, Kobus Barnard, Nando de Freitas, David Forsyth and Michael I. Jordan. "Object Recognition as Machine Translation: Learning a Lexicon for a Fixed Image Vocabulary." *European Conference on Computer Vision*, 2002.
- Peter Carbonetto, Nando de Freitas, Paul Gustafson and Natalie Thompson. "Bayesian Feature Weighting for Unsupervised Learning, with Application to Object Recognition." *Artificial Intelligence and Statistics*, 2003.
- Peter Carbonetto and Nando de Freitas. "Why Can't José Read? The Problem of Learning Semantic Associations in a Robot Environment." *Workshop on Learning Word Meaning from Non-Linguistic Data in the Human Language Technology Conference,* 2003.

My Masters thesis should be done sometime in the near future, which will be the definitive reference for all this stuff.

There's lots of stuff in this package, so it is worth taking the time to go through each part step by step, as we will do in succeeding sections. All the code is written in Matlab. More specifically, using version 6.1 on RedHat Linux platform version 7.3. You might encounter problems if you run the program a different operating system or a different version of Matlab. Matlab has its drawbacks, I find ease of debugging code makes it worthwhile in the end. I hope you do too. And of course, if you find any problems with this package, you may contact me at `pcarbo@cs.ubc.ca`.

Before going into the different sections, let me explain how the code is organized. Additionally, let me stress that you should read this entire document before using my code. It's really not very long, and I assure you that the things I say will help you in the long run. There is a total of five "subpackages", each identified by its own subdirectory. These subdirectories are `segmentimg`, `createlabels`, `createdatasets`, `translation` and `resultsbrowser`. There's also a subdirectory called `general`, and that contains multi-purpose code used by all the subpackages. I won't get into that code in any amount of detail, but it contains some routines I wrote for parsing strings, manipulating matrices and cells, creating a caches for the user interfaces, and loading and saving data. Finally, there are some Matlab files (the ones that end in the `.m` extension) in the main package directory. Those are there either because they didn't fit in any of the subdirectories or because they are examples showing how I used the functions on my computer, and can serve as a concrete starting point.

Let's create a story for all this code. Say you have a bunch of images in a single directory. First what you want to do is segment the images and create the sets of features for each segment. The function `segmentimg/create_segment_data` will do that all for you. After waiting a bit of time for Matlab to process all the images, you now have loads of information contained in one directory in a format subsequent functions will be able to understand. Next, what you'll want to do is create the labels for each of these images and provide the manual correspondences for the ground truth, so we can test your models properly. Fortunately, I've created an elegant interface (if I must say so myself) for doing just that, `createlabels/createlabels`. Once you've created proper labels for all the images, it is time to put everything together and create a data set using `createdatasets/create_datasets`. Next, use

`translation/evaluate_model` to train and translate the test and training sets for your model(s). Finally, you'll want to inspect the results to see how your model performed on the data set you built. You can do this by plotting the average error measures for different models using `resultsbrowser/compare_models` or you can view individual results by starting up the interface `resultsbrowser/imagebrowser`.

One caveat regarding the functions described in later sections: all directory specifications should be *absolute*, not relative. In other words, if you are using Matlab on a UNIX-based platform, the directories should all start with the "/".

Also, throughout this package I use the words "blob", "segment" and "patch" and they all mean the same thing: a contiguous region of an image.

Creating segment data

If you look in the subdirectory `segmentimg` you will see a lot of functions, but the only one you really need is `create_segment_data`. This function segments the images and produces features for each image segment.

First, you want to set up a directory and one of the subdirectories, usually called `clrimages`. A function call will look something like this if you want to segment the images using Normalized Cuts.

```
datadir  = '/project/imagetrans/data/imagesA';
ncuts    = '/bin/ncuts';
features = [1 3:6];
options  = {'ncuts', [ ], [ ], ncuts, ...
            [ ], features};
create_segment_data(datadir, options);
```

Otherwise, if you would like to use the grid segmentation instead, the list of Matlab commands would look like the following:

```
datadir  = '/project/imagetrans/data/imagesA';
ncuts    = '/bin/ncuts';
features = [1 3:6];
patchsz  = 24;
crop     = 6;
options  = {'grid', patchsz, crop, [ ], ...
            [ ], features};
create_segment_data(datadir, options);
```

For more information, type `help create_segment_data` in the Matlab prompt.

The set of features you can compute are listed in file `features.txt`. Each row lists the name of the feature, the number of dimensions it adds to the patch information vectors, and the Matlab file that is called to compute the information for a single patch. The vector `features` shown in the above code refers to the rows of this table. In order to customize the set of features, you can add a row to the `features.txt` file. (For example, texture is currently absent from the set of possible features to compute, so you may want to add it.) To create your own Matlab feature-computing function, you need to follow the following template:

```
% COMPUTE_MY_FEATURE
% Comments go here.
function f = compute_my_feature(img, nimg, ...
                                labimg, segment)
   % Function body goes here.
```

`img` is a $H \times W \times 3$ matrix of RGB colour-space values where H is the height of the image and W is the width of the image. `img` is the same as `nimg`, only the values are normalized by the luminance of the image. `labimg` is a $H \times W \times 3$ matrix of entries in the CIE-Lab colour space. `segment` is a $H \times W$ matrix, where entry $(h, w) = 1$ if and only if that pixel is in the segment. Otherwise, the entry is 0. The return value is a $1 \times F$ vector of feature values, where F is the number of features, and should correspond to the second column of your new entry in the table `features.txt`.

Segmenting images creates a lot of data in the directory specified by `datadir`. I won't into complete detail, but I will explain briefly what the most important files are. It creates some images in the subdirectories `segimages` and `blobimages`, but these are only used for the graphical user interfaces you will use later on. Matlab files describing the image segments are contained in the subdirectory `segments`.

Open the text file `image_index`. At the top is the list of subdirectories where the auxiliary information is stored. After that, there is a list of all the images that were segmented. If you would like to add more images to this set later on, it is simply a matter of copying them to the `clrimages` subdirectory and re-running `create_segment_data` since it keeps track of the images it has already segmented. If you open the text file `blob_features`, you will first see the set of feature names used. Below in the file is the computed set of features for each patch in each image. As you should observe, this can be a very large file. Finally, the `adjacencies` file contains a single adjacency matrix for each image describing how the patches are spatially organized; a 0 means the patches are not adjacent, a 1 and 2 encode the above and below relations, and 3 means the two patches are next to each other.

Labeling the data

Once the images have been segmented and the feature and adjacency information has been created to your satisfaction, the next step is to manually annotate and label the images. I have created a Matlab interface called `createlabels/createlabels`. Calling the interface would look pretty much like this:

```
datadir = '/project/imagetrans/data/imagesA';
createlabels(datadir);
```

Let's take a look at how the `createlabels` interface works. I will occasionally refer you to Figure 1. The original is displayed on the bottom-left of the interface, and the segmented version on the top-left. To add a word to the image caption, type it into the **word** text box and click on the **add/view** button. If the word does not already exist in the vocabulary, it will automatically be added. You can also add a word if it already exists in the vocabulary by highlighting it, then clicking on **add/view**. Delete a word from the image with the **delete** button. Note that you can add or delete multiple words, either by selecting them in the vocabulary or typing them in the text box. Once a word is highlighted in the vocabulary, you can start labeling the individual words by clicking on the top-right image. The patches that are annotated with the highlighted word are shown in red. If you try to add a word that already exists in the image label, it will simply display the patches that are annotated by that word.

The four buttons below the vocabulary operate on it. The purpose of most of the buttons is pretty self-explanatory. If you have two words in the vocabulary that are synonyms (e.g. "sea" and "water"), you can merge them first by selecting two or more words in the vocabulary and then clicking on the **merge** button.

Click on the **forward** and **back** buttons to scroll through the set of images. Also, enter an image number in the box in between the two buttons to go to an arbitrary document in the set of images. Once in awhile, and definitely before quitting, you should click on the **save** button to save the labellings to disk.

The window is resizable.

Recovering labels

Say, for instance, you have the situation in which you laboriously labeled and annotated a large collection of images. Then, you decided that to improve the segmentation of the images, so you re-segmented them. Ordinarily, you would have to re-label all the data. You can use the `recoup_labels` function in the `createlabels` directory to recover the previously-labeled data. This will not result in a perfect annotation, but it will be pretty good. For certain segmentations, you may not have to re-annotate the patches at all.

Creating data sets

Once you've labeled the data to your satisfaction, you can now go about creating a data set by merging several sets of labeled images. The function you will be using is `createdatasets/create_datasets`. Calling the function will look something like this:

```
datadir1    = '/project/imagetrans/data/imagesA';
datadir2    = '/project/imagetrans/data/imagesB';
resultdir   = '/project/imagetrans/data/set1';
datalabels  = {'training' 'test'};
proportions = [3 1];
seed        = 2067;
create_datasets(datadirs, resultdir, ...
   datalabels, proportions, 0, seed);
```

For more information, type `help create_datasets` in the Matlab prompt.

Once you've run `create_datasets`, you will find a lot of new files that have appeared in the result directory. The file `specs` contains some information pertinent to the whole data set, including the names of the features, the names of the sets and information the graphical user interfaces use to locate the image data. In addition, each training and test set is stored in a separate subdirectory. Each directory contains the files: `adjacencies`, `blob_words`, `blobs`, `image_blobs`, `image_words`, `images` and `words`. I will give you a brief tour of what information each of these files contains.

The `words` file is a list of all the words in that data set. Usually one refers to words by their index, which is simply the line number in this file.

The list of image names for the dataset are found in the `images`. The first entry on each line refers to the image set number, and the second is the name of the image. To find out exactly what file each document points to, you need to look up the directory pathname in the `specs` file, tack on the name of the image as listed in the `images` file and then add the image suffix at the end, again described in the `specs` file.

Each line of `image_words` is a label for document n, where n in the line number. Each positive number corresponds to a line number in the `words` file. The length of a document's label is specified by the number of positive numbers.

Each row in the `image_blobs` is a list of all the blobs (or patches) in the document, just as in the file `image_words`.

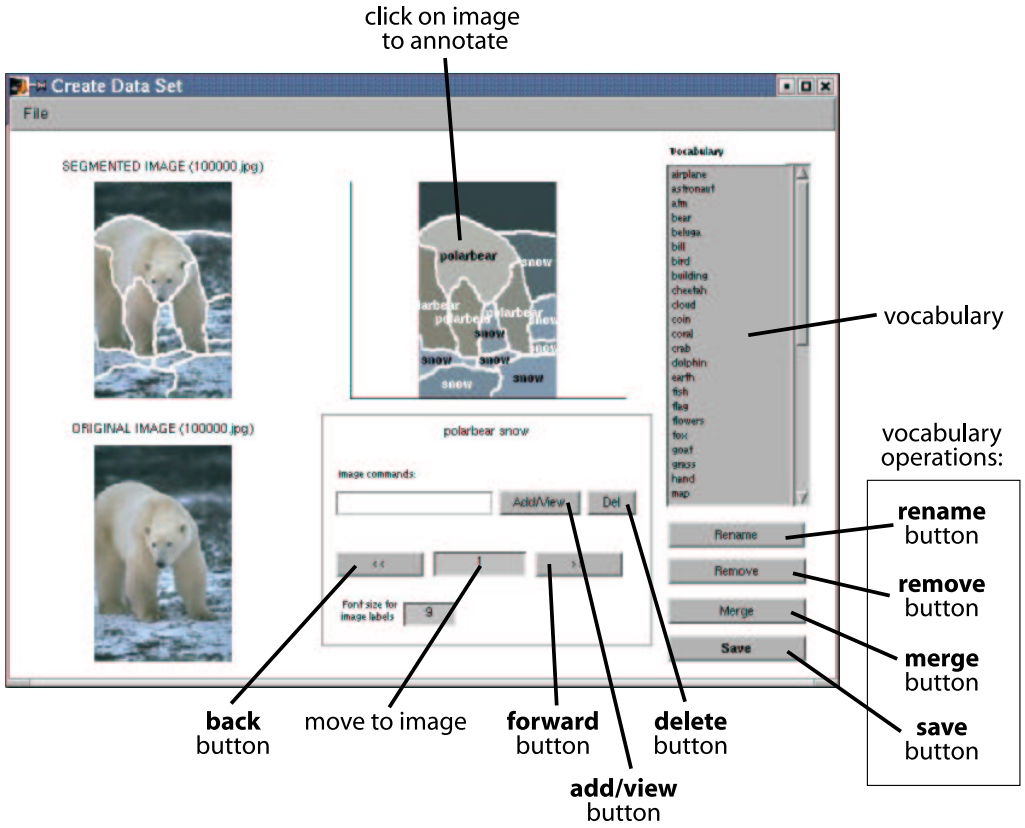

Figure 1: The `createlabels` interface.

Each number refers to a line number in `blobs`, to be described shortly. To figure out how many patches are in a single document, simply count the number of non-zero entries.

`blobs` contains the feature information for the blobs in the documents. Each row is a single feature value. To find out what these features actually mean, look in the `specs` file.

There's more information about the blobs contained in the file `adjacencies`. Each entry in this document is a 2-dimensional matrix, where entry (i, j) is positive if the ith and jth patches in the document are adjacent. More specifically, a 1 means that patch i is next to patch j, 2 means i is below j, and 3 represents the above relation. Each document takes up a number of lines equal to the number of blobs, which you can find from the `image_blobs` file.

Lastly, we need to talk about the `blob_words` file. Like the adjacencies file, each document takes up a number of lines corresponding to the number of patches. Each line corresponds to a single patch in a single document, and lists zero, one or more possible correct annotations, as deemed by the annotator. Each non-zero entry corresponds to a line number in the `words` file.

Training your model

Okay, now we are getting to the meat of the **imagetrans** package here. The method you will be using is `translation/evaluate_model`. There are a lot of issues to discuss with respect to this function, but first I will show a typical script for training a model:

```
datadir   = '/project/imagetrans/data/set1';
modeldir  = '/project/imagetrans/data/models';
model     = 'modelA';
numtrials = 16;
evaluate_model(datadir, modeldir, model, ...
               1:numtrials);
```

In this specific situation, what `evaluate_model` does is loads the data located in the directory `/project/imagetrans/data/set1`, loads the specifications for the model contained in a text file `/project/imagetrans/data/models/modelA.txt`, trains the model using the data on trials 1 to 16, and stores the results in

`/project/imagetrans/data/set1/results/modelA`. Most of the results are stored in subdirectories, one for each trial. Note that all model specification files must end with `.txt`. The trial numbers must be specified as a vector; you don't necessarily have to start with trial 1. This is useful if you want to run several of the trials in parallel on different machines.

So far, I've told you the minimal information to evaluate a model. However, if you want to write the specifications for your own model or program your own training algorithm in Matlab, you need to know more, so read on.

Say you would like to create your own model. First, look at the file `translation/models.txt`. Each set of five lines, including the blank line, encodes the information needed to run a particular model type. The first line is the name of the model. The second, third and fourth lines are names of Matlab functions for training the model, writing the learned model parameters to disk, and translating a set of documents using the trained model. To create your model, the first thing you need to do is create an entry at the end of the table, since the function `evaluate_model` will refer to it.

Once that's done, your next task to write the functions corresponding to the function names listed in the `models.txt` table, using the following templates:

```
% MYMODEL_TRAIN
% Comments go here.
function [model, val] = mymodel_train(B, W, ...
                          A, M, L, numwords)
  % Function body goes here.

% MYMODEL_WRITE
% Comments go here.
function mymodel_write(d, model)
  % Function body goes here.

% MYMODEL_TRANS
% Comments go here.
function t = mymodel_trans(B, M, A, model)
  % Function body goes here.
```

The function parameter `B` is a $F \times B \times N$ matrix where F is the number of features, B is the maximum number of blobs in an document or image, N is the total number of documents, and each entry (f, b, n) contains a single feature value for a blob. `W` is an $N \times W$ matrix of word indexes where W is the maximum size of an image label; i.e. the maximum number of words in a document. `A` is a cell array of adjacency matrices of length N. Each adjacency matrix is of dimension $B_n \times B_n$, where B_n is the number of blobs in document n and entry (i, j) describes the spatial relation between two blobs with indexes i and j. For more information on the adjacency matrix, see the description in the section "Creating data sets". `M` is an $N \times 1$ matrix of blob counts and `L` is a $N \times 1$ matrix of word counts or label sizes. `numwords` is the total number of word tokens in the data set. In addition to these parameters, the function `mymodel_train` can also take a set of model-specific parameters. We will discuss model specification files in more detail below.

The model training function has two return values. The first, `model`, is a struct containing any information pertinent to image translation. You can fill this with information to your discretion. `val` is a number describing how good the result of training is, which is needed when the user runs several trials and wants to make an educated guess about which is the best training run. For example, this could be the computed joint log likelihood or posterior.

The parameters for `mymodel_write` are `d`, the directory to write to, and `model`, the value returned from the training function described above.

The function parameters for the translation function are equivalent to those described above. The return value `t` is a $V \times B \times N$ matrix where V is the total number of word tokens considered by the model, B is the maximum number of blobs in a single image and N is the number of images. Each entry (w, b, n) is the probability that blob number b in document n is annotated with word index w. Since they are probabilities, all the words for each blob and image should add to 1.

Let's say you've written the functions for your new model. To test it, you need to create a model specification file. Going along with the sample script written above, let's call it `modelA.txt`. The text file will look something like this:

```
Model label:          "Model A"
Model name:           "mymodel"
Features:             3 8:13
Blob area threshold:  0.01
Num blobs per doc:    "all"
Num restarts:         6
Num faulty starts:    50
```

All the lines listed above are required for all models. In other words, these are model-generic parameters. The first line is simply a tag to describe the model, and will be used for display in graphs and so on. The model name must be the same as the name listed in the `models.txt` file. The third line is the dimensions of the set of blob features you would like to use for training (to see what they are, look at the `specs` file in the respective data set). If you would like to use all the features, set it to `"all"`. The `Blob area threshold` must be a number between 0 and 1. If the area ratio of a blob is less than this number, we remove it from the set of blobs. To keep all the blobs,

set it to 0 or type "none". If a "C" is added to the end of this value and the whole thing is surrounded with double quotes, the blobs will also be thresholded by area during the test phase. Num blobs per doc is the maximum number of blobs in a document, and we remove any that exceed this number. If you want to keep all the blobs, specify "all". The next line is the number of times to train the model, keeping only the best one. Some algorithms fail under rare initial conditions. This is especially the case when we are sampling from unstable distributions. If so, you might want to set Num faulty starts to a large value so that the program recovers from a failure. Otherwise, set this value to 1.

In this package, there is already a set of models you can try out. We will now proceed to describe how to use them by listing sample model specification files. The settings that I use below tended to lead to good results.

```
Model label:          "dML1"
Model name:           "Model 1 discrete ML"
Features:             "all"
Blob area threshold: 0.01
Num blobs per doc:    "all"
Num restarts:         3
Num faulty starts:    1
KM iterations:        24    % For K-Means
EM iterations:        24
EM tolerance:         0.1
```

```
Model label:          "dML1O"
Model name:           "Model 1 discrete ML online"
Features:             "all"
Blob area threshold: 0.01
Num blobs per doc:    "all"
Num restarts:         3
Num faulty starts:    1
KM iterations:        24    % For K-Means
EM iterations:        4
Learning schedule a: 0.98  % Scaled by number of
Learning schedule b: 0.05  % documents
```

```
Model label:          "gML1"
Model name:           "Model 1 Gaussian ML"
Features:             "all"
Blob area threshold: 0.01
Num blobs per doc:    "all"
Num restarts:         3
Num faulty starts:    50
KM iterations:        "n/a"  % For K-Means
EM iterations:        24
EM tolerance:         0.1
Diagonal covariance: "no"
Random init:          "yes"  % If "yes", we don't
                             % use K-Means.
```

```
Model label:          "gML1O"
Model name:           "Model 1 Gaussian ML online"
Features:             "all"
Blob area threshold: 0.01
Num blobs per doc:    "all"
Num restarts:         3
Num faulty starts:    50
KM iterations:        "n/a" % For K-Means
EM iterations:        24
Learning schedule a: 0.98  % Scaled by number of
Learning schedule b: 0 0.1 % documents. The first
                        % b corresponds to means,
                        % the second to variances.
Diagonal covariance: "no"
Random init:          "yes" % If "yes", we don't
                            % use K-Means.
```

```
Model label:          "gMAP1"
Model name:           "Model 1 Gaussian MAP"
Features:             3 8:13
Blob area threshold: 0.01
Num blobs per doc:    "all"
Num restarts:         3
Num faulty starts:    1
KM iterations:        "n/a" % For K-Means
EM iterations:        20
EM tolerance:         0.1
alpha:                10    % Prior on Sigma
a:                    0.8   % Prior on Tau
b:                    0.5   % Prior on Tau
Random init:          "yes" % If "yes", we don't
                            % use K-Means.
```

```
Model label:          "gMAP1MRF"
Model name:           "Model 1 MRF Gaussian MAP"
Features:             "all"
Blob area threshold: 0.01
Num blobs per doc:    "all"
Num restarts:         1
Num faulty starts:    50
KM iterations:        "n/a" % For K-Means
EM iterations:        24
EM tolerance:         0.1
alpha:                10    % Prior on Sigma
a:                    0.8   % Prior on Tau
b:                    0.5   % Prior on Tau
phi:                  0.01  % Prior on psi
sum-product:          "no"  % Use mean.
                % Otherwise, use argmax.
Random init:          "yes" % If "yes", we don't
                            % use K-Means.
```

Okay. That's really all you need to know for training and testing models.

Viewing results

There are several ways you can go about inspecting the evaluation of your model on a particular data set. The functions you will be using most often in the resultsbrowser subdirectory are imagebrowser, compare_models, compile_stats and show_stats.

One, you can inspect the evaluation anecdotally using the resultsbrowser/imagebrowser interface. The function

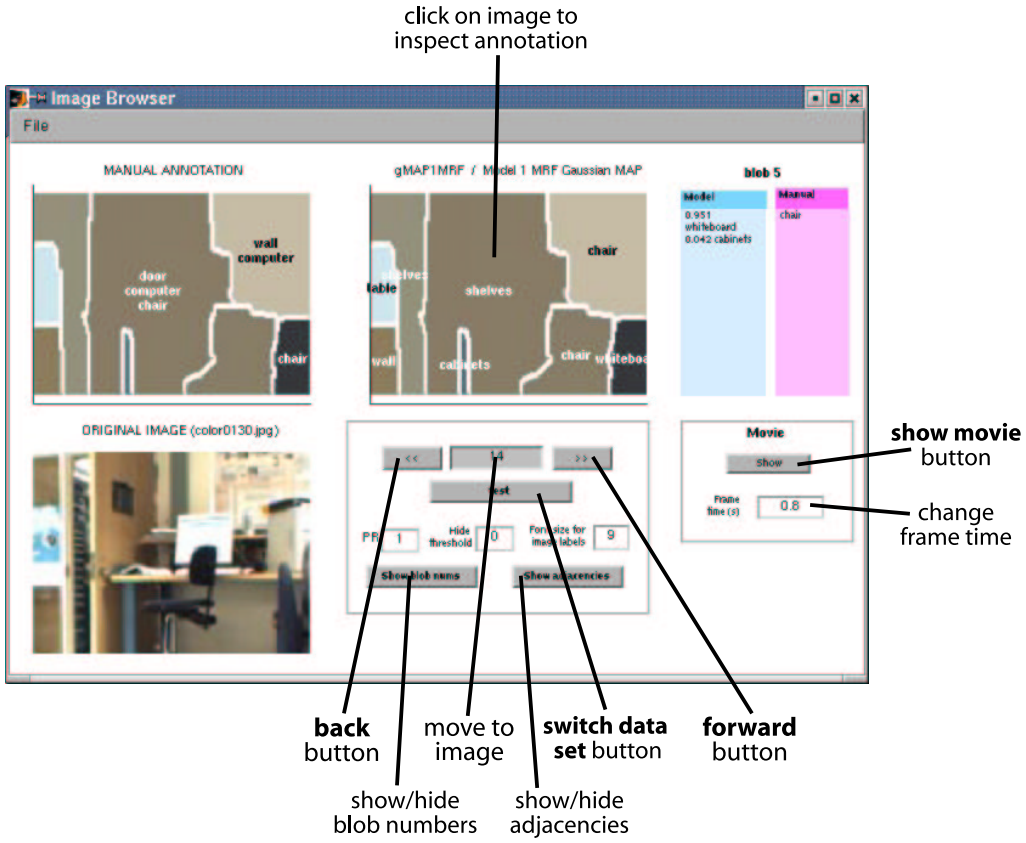

Figure 2: The `imagebrowser` interface.

call is simply `imagebrowser(datadir, modelname)`. You can enter an empty matrix for `modelname` if you just want to view the manual annotations. For more information on the function call, check out the function help.

The `imagebrowser` interface is shown in Figure 2. Click on the **forward** and **back** buttons to scroll in the images. Also, enter an image number in the box in between the two buttons to go to an arbitrary document in the set of images. Also, there is a `show movie` feature which steps through the images automatically. This is useful if the images are from sequentially ordered navigation data. To change the data set, click on the `switch data set` button. The current data set is displayed on the button.

Click on a blob in the top-right image to inspect an annotation more closely. The manual and model annotations (with their translation probabilities) are shown to the right. The text box `PR` changes how many words are displayed for each blob. The default is 1. If the image is getting too cluttered with labels, you can hide the labels for some of the smaller patches by increasing the `hide threshold` number, to a maximum of 1 (which hides all the annotations). Alternatively, you can change the font size.

There's a button to display the blob numbers along with the annotations. The `show adjacencies` button replaces the annotations with graph, where the edges represent adjacencies between blobs. By clicking on a blob while adjacencies are displayed, you can get more information about spatial relations in the table on the right.

Next we look at the function `compare_models`. A script to call this function would look like this:

```
datadir    = '/project/imagetrans/data/set1';
modelnames = {'modelA' 'modelB' 'modelC'};
test       = 'pon';
pr         = 1;
boxplot    = 1;
options    = {[ ] boxplot};
compare_models(datadir, modelnames, test, ...
               options, pr);
```

For more information, consult the help. The parameter that requires immediate explanation is `test`. It specifies which test to use. Currently, there are three possible tests, each with a variable set of parameters. The parameters for the tests are inputted to `compare_models` after the fourth

Test	Description
prs	Error is one minus the probability that the blob is annotated with the correct word. The error is averaged over the number of blobs in the document, and then over the number of documents. There are no parameters.
prn	Error is 0 of the blob is annotated the correct word on one of the top n most probably predicted words, where n is the test parameter. Otherwise, it is 0. The error is averaged over the number of blobs in the document, and then over the number of documents.
pon	The difference with this measure compared to 'prn' is that we average first over the blobs annotated with a specific word, then over the number of words in the label, and then finally over the number of documents. n is the parameter.
pos	Error is one minus the probability that the blob is annotated with the correct word. The error is averaged first over the blobs annotated with a specific word, then over the number of words in the label, and then finally over the number of documents.

Figure 3: A description of the tests used by the `compare_models` function.

parameter. In this case, `pr` is the single parameter used by the test `pon`. The tests that are available are listed in Figure 3. It is possible to create your own test function using the following template:

```
% RUN_TEST_MYTEST
% Comments go here.
err = run_test_mytest(manual_words, ...
         manual_blobwords, model_words, ...
         model_blobwords)
  % Function body goes here.
```

If your test is named "mytest", then the file storing the function should me named `run_test_mytest.m`. `manual_words` and `model_words` are cell arrays containing the word tokens for the manual annotator and the model annotator, respectively. `manual_blobwords` is a $W_1 \times B \times N$ matrix, where W_1 is the total number of word tokens in `manual_words`, B is the maximum number of blobs in an image, N is the number of images and each entry encodes the probability of translation. `model_blobwords` is a $W_2 \times B \times N$ matrix, were W_2 is the total number of word tokens in `model_words`. After these four parameters, you can add test-specific parameters as you desire.

Finally, we'll look at the functions `compile_stats`, `show_stats` and some others. With these functions, you can look into more detail the results of your model.

Use `show_stats(stats,trial)` to view the stats returned by `compile_stats`. The parameter `trial` specifies which trial number to display, or `0` for the averaged results. `show_stats_latex` does the same thing, only it displays the tables in a format amenable to Latex documents.

Use `show_psi(stats)` to display the parameter ψ learned from some models. Use `view_taus(datadir, modelname)` to display the variable weighting parameter τ, which is useful for viewing the learned relative importance of the features. Use the Matlab help for more information.