CS 221
Object Recognition and Tracking

You may work in teams of up to 3 students for this project. Some important dates:

- You must form a team by **Friday, October 9**, 11:59pm. Send us an email with your team details, as described in Section 3. If you want to define your own project, the project proposal is also due on this date, but you should contact us *well in advance* to make sure what you have in mind meets the proposal requirements.
- The deadline for the milestone is **Tuesday, October 20**, 11:59pm.
- The final program submission deadline is **Thursday, December 10 at 12:15pm** (no late days).
- The final project competition will be on **Friday, December 11** from 12:15-3:15pm.
- The written report is due on **Friday, December 11**, 11:59pm. (no late days)

You may use late days on the milestone,[1] but not the program submission or the writeup submission, to allow us to run the competition and complete grading in time.

1 General Description

This handout describes one of the two challenge problems for this course. These challenge problems have been picked to be quite open-ended and you are encouraged to experiment with different ideas, algorithms, functions, and so forth. Indeed, those teams that do explore will most probably do better in the competition.

Honor Code: You may consult any papers, books, or online references for ideas that you may want to incorporate into your strategy or learning algorithm, so long as you *clearly cite your sources* in your code and your paper. However, under no circumstances may you look at another student's code or incorporate their code (either from this year or a previous year) into your project.

2 Challenge problem 2: Object recognition and tracking

In this challenge problem, you will build an object recognition and tracking program. The program should identify and label all occurences of 5 objects (mug, stapler, keyboard, clock, and scissors) in a given video clip. You will be given several labeled still images of each of these objects as training data (similar to those in Table 1), and several labeled short videos to run experiments on. During the final competition, your program will be evaluated on similar but unseen videos, as described in Section 8.1.

[1] Please note that late days taken by your team will count against *each team member's* supply of late days; for example, if you take 2 late days then each member of your team will be charged 2 late days.

Table 1: Sample training images for a mug.

3 How to Start

Don't worry if, at the moment, you don't understand all of the concepts mentioned in this handout. Most of the new topics will be covered in class.

- Review this document and the associated code. The code is in directory `/afs/ir/class/cs221/vision/code/`
- Pick a name for your team and send an email to `cs221qa@cs.stanford.edu` in the following format by **Friday, October 9** at 11:59pm:

```
To:cs221qa@cs.stanford.edu
Subject: Vision Project Teams:Cardinals

Team Name: Cardinals
Team Member1: Joe Smith <jsmith@cs.stanford.edu>
Team Member2: Jane Doe <jdoe@cs.stanford.edu>
...
```

- Make sure you can access and use the software supplied to you.
- Come to the information session (date and location TBA) where we will describe the code infrastructure and useful OpenCV library functions, or watch it online.
- Begin working on the tasks for the milestone. While you are free to pursue whatever techniques you want for the final submission, doing the milestone will get you on the right path and give you ideas on how to approach the problem.

4 Outline Of Tasks For You To Do

Since the project is quite large we have created a milestone to get you started. We have designed the milestone so it will cause as little overhead as possible for you. It just means that you have to complete some of the work by a specific date.

4.1 Milestone

For the milestone, we will focus on just one object: the mug. You will be given a number of still images of that object to train a classifier. You will also be given a list of fragment-based features which you will use as the features in your classifier (see section 6 for details). Using these features, you will implement a logistic regression classifier that detects the object in a given new image. Given a test video, your program should then apply the learned classifier to independently classify each *candidate region* in each video frame. In other words, you will test your classifier on various sections of the image (ones at least 64×64 pixels in size) and determine whether the section of the image contains the object in question.

Below we will go into greater detail over the tasks mentioned above, giving some ideas for implementing them.

4.2 Final Submission

For the final submission, your program should attempt to identify all five objects: mug, stapler, keyboard, clock, and scissors. One possible way to build such a program is to implement a separate classifier to identify each of the five objects; or, you might implement a single classifier that attempts to identify all five; or do anything in-between (such as build separate classifiers, but have them all share features). You must also improve the accuracy of your classifier by implementing several extensions.

You should come up with your own extensions. You're also welcome to discuss them with us during office hours or over e-mail. Here are some possible ideas for extensions to get you started:

- **Alternate Features** You don't need to remain committed to the fragment-based features for your final submission. Perhaps taking ideas from the vision literature, you could augment your feature vector with other types of features or replace them altogether with different image features. Some of these papers might give you ideas.
  - Navneet Dalal and Bill Triggs, "Histograms of Oriented Gradients for Human Detection." *CVPR '05* - Volume 1, pp. 886-893
  - Lior Wolf Thomas Serre and Tomaso Poggio, "A new biologically motivated framework for robust object recognition," *Al Memo 2004-026*, November 2004.
  - David G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, 60, 2 (2004), pp. 91-110.
- **Custom Features** Instead of using the features given to you or ones from the literature, try to design your own features to use in the classifier. For example, is there other image pre-processing (such as normalizing contrast, applying an edge detector, color features, etc.) that helps?
- **Temporal reasoning** Instead of analyzing each 10th frame (see Section 5.2.1) image individually, try to use other frames in the video to help eliminate false positives and false negatives (for example, if the previous image frame contained a mug, it is very likely that the next image frame contains that same mug). One way to capture the inter-frame dependency would be to include it in a probabilistic model, and perform inference on the model, using techniques such as a *particle filter*. Note however the running time constraint (Section 8.3).

- **Joint classification** Devise a clever way to combine the classifier's output on different candidate regions (search online for "non-maximum suppression"), or for combining information from the different object classifiers.
- **Create Your Own** Try to come up with your own method for improving the performance of your classifier, be creative!

You must submit the code you have written and an executable file. Your code should be well written, modular, and readable. You should put comments in the important procedures of your code. In addition to the code, you must also run some experiments with your program and submit a report. The report should explain what you did in the project and analyze your experiments. It should include the following parts:

- Description of the important data structures and procedures of the code you wrote. This needs to be a brief, general description. You should also mention any special tricks you used to get speedups.
- Describe and justify the extensions you implemented.
- Any assumptions you made.
- Present experimental results. You should feel free to devise your own experiments to evaluate interesting aspects of your classifier, but here are some ideas for experiments you can conduct:
  - Design a baseline classifier that uses basically the code you submitted for the milestone (but extended to handle multiple object types). How good is this baseline?
  - How much benefit do you get from different components of your submission? Or, put another way, if you remove one of the components from the final submission, how is the final performance affected? (Such studies are typically called *ablation analyses*.)
  - What are the strengths and weaknesses of your submission? For example: is your classifier very good at finding mugs, but poor at finding staplers? A standard way to devise better learning algorithms is to analyze such errors systematically.

We have given an outline of the project and a few suggestions on how to improve it. Note that this is by no means a complete list of things you can do and you are encouraged to explore.

4.3 Grading

This project is worth 40% of the final grade. The project will be marked out of 100 points. The work handed in for the milestone will account for 20 points and the final submission will account for the remaining 80 points.

4.3.1 Milestone

- **Full Credit (20 points):** you have correctly implemented the tasks described in Section 4.1. For details see sections 6 and 7. Your code executes without errors on our test videos and performs comparably to our implementation of the milestone. We expect an F_1 score of 0.25 or higher (see 8.2 for an explanation of F_1 score).

- **Conditional Full Credit (20 points):** you have implemented the majority of the required tasks however there are a few flaws. You will receive full credit if these are fixed by the final submission.
- **Partial Credit (lose some of the 20 points):** significant parts of the milestone that were supposed to be implemented were not implemented or do not work.

4.3.2 Final Submission

You must demonstrate significant progress beyond your milestone submission to obtain full credit for the final submission. We recommend that you implement at least three sizeable extensions beyond the milestone; feel free to talk to us during office hours if you have questions or want to discuss ideas for extensions.

Your grade will depend primarily on three factors, the first two being the most important. The first is the quality of your project as a whole. Does your program implement the required functionality? Did you try a few interesting extensions?

The second is the quality of your report. Some issues that we will be looking at are:

- Are your extensions described clearly?
- Did you justify your design decisions?
- Did you run interesting experiments?
- Did you analyze your results?

The third is the quality of your code. We may deduct points for code which is badly written or nonmodular.

Although all three factors contribute to your grade, our primary insight into your submission comes from the written report. **In order to receive full credit for the project, your report must describe your extensions in detail and analyze their effectiveness.** View the writeup as a scientific paper in which you must justify your work. Perform experiments such as running your program both with and without each extension, and present the results using charts or graphs. It's fine if the experiments show that your extensions did not improve performance, as long as you demonstrate that you explored interesting and reasonable strategies in your program and analyzed the results of your experiments.

Extra credit: Extra credit will be given to the top two teams. Each member of the winning team will get 3 points of extra credit, and each member of the runner-up team will get 1.5 points of extra credit. The points will be added to the final course grade after the curve has been decided, so they will not affect other students. In case of a tie, the point values will be split evenly among the teams.

In addition, we will award extra credit points to projects that have significant and creative additional extensions.

5 Implementation Issues

5.1 Computing resources

The vision project depends on the OpenCV library. Your starter code should automatically link against the class copy of the OpenCV library. Unfortunately, because the different clusters

have different processor architectures and different versions of the C compiler, the class copy of OpenCV will not work on the `cardinal` cluster.

You are free to develop your project on any of the `myth`, `corn`, or `pod` machines, or on your own machine. However, be sure that your code compiles and runs without error on the `myth` machines prior to submission. We will use the `myth` machines to run the final competition.

5.2 Code Infrastructure

Copy over all the files from `/afs/ir/class/cs221/vision/code/` into your code directory. You are free to copy over the data if you like. The code makes use of the Open Computer Vision Library (OpenCV). There are three executable files that are created when you run `make`. Before making the executables you should set your library path to include the OpenCV libraries (the following is a one-line command):

```
setenv LD_LIBRARY_PATH
    ${LD_LIBRARY_PATH}:/afs/ir/class/cs221/stairvision/external/opencv/lib
```

If you get an error message saying, "LD_LIBRARY_PATH: Undefined variable," then simplify the command to:

```
setenv LD_LIBRARY_PATH /afs/ir/class/cs221/stairvision/external/opencv/lib
```

To avoid needing to run this command again every time you log in, you may want to add it to your .cshrc.user file located in your home directory.

After you have compiled the code you should have the following executable files:

train takes a directory name as input and calls the `Classifier::train` method with a list of image filenames for each object class on which to train. The image class is determined by the subdirectory name in which the images are found. The optional argument `-c <filename>` will cause **train** to save the trained parameters in the given file (by calling `Classifier::saveState`).

test takes the path to a directory full of video frames as input and first calls the `Classifier::loadState` method to initialize the classifier and then repeatedly calls the `Classifier::run` method on each video frame. The program displays the classification results (and optionally ground truth labels). Use the `-c <filename>` argument to control which file is passed to `Classifier::loadState`.

evaluate compares the output from your classifier to the ground truth labels. It displays information that may help you quantify the kinds of mistakes that your classifer makes, as well as your classifier's overall score.

Do not modify `test.cpp` or `replay.cpp`. You are free to modify all other code. In particular you should implement the public methods in `Classifier`.

In your code directory, you should be able to compile using `make` , and then you should be able to run the following command:

```
./test -g /afs/ir/class/cs221/vision/data/easy.xml
          /afs/ir/class/cs221/vision/data/easy
```

This runs the test command on the video `easy` with superimposed ground truth labels from `easy.xml`. (We use directories full of images rather than .avi files to represent video for this assignment, so that students working outside of the `myth` cluster don't need to compile OpenCV with .avi file support.)
You should also go over the comments inside the `classifier.cpp` file. The `run` function is currently configured to output random rectangles for each frame.

5.2.1 Evaluation

You will only be evaluated on every 10th frame of the video. This means that even though you are free to use information from all frames (e.g., for tracking the objects) only the detection results you output on frames $5, 15, 25, \ldots$ will actually be scored.

5.2.2 Information session

We will have a special information session where we will discuss the code infrastructure and useful OpenCV methods (date and location TBA).

5.2.3 The Graphical User Interface

The test program will display each video frame that is passed to your classifier. You can disable the GUI with the option -x. The program also shows your classifier results as green rectangles. You can supply a file of ground truth labels with the -g options. This will display ground truth as red rectangles.

5.3 Etiquette

Parts of this project may involve intensive computation. The Leland machines are a shared resource for all of Stanford which should not be abused. To be fair and limit the load on the cluster, we ask that each group use a maximum of six processes total on at most three machines. If you are running a long experiment, you should nice the processes, like this:

```
% nice +10 myprog parameters
```

Exceeding the above recommendations will be considered a violation of class rules.

5.4 Submitting

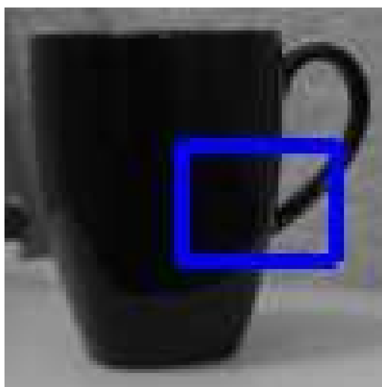
To submit your program, make sure you are in the directory containing all of your code, and execute the script available at `/afs/ir/class/cs221/bin/submit`. From there, follow the directions. You are allowed to submit multiple times, but only your most recent submission will be looked at. In addition, late days for the milestone will be assessed based on the submission time of your last submission. No late days are allowed for the final program submission.

6 Localized fragment-based features

6.1 Overview

For the milestone, you will implement a type of image features that have been used to obtain state-of-the-art results for object classification[2]. Each feature is defined by an image fragment and a location. In order to understand how these features work, there are three basic ideas:

1. **Feature dictionary.** Consider a typical training image from which we've cropped out a fragment (shown in blue):



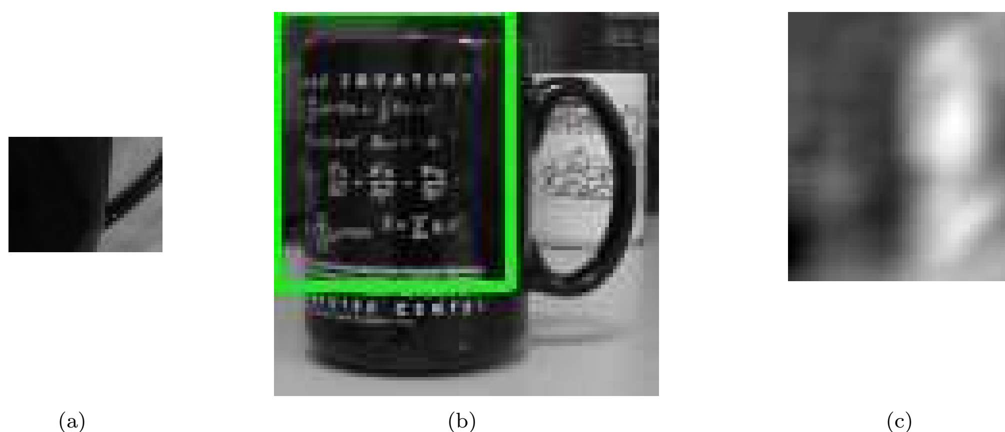
We build a *feature dictionary* by repeatedly cropping out such fragments at random locations on the positive training examples and saving the results. We store both the fragments themselves (commonly refered to as the *templates*) as well the locations of their upper lefthand corner within the original training image. Section 6.1.1 will cover this process in more detail. For the milestone we provide a dictionary for you to use, so you don't have to worry about finding informative fragments.

2. **Response image.** Given a new image, we can then compute the *response* of each fragment on this image by "sliding" the fragment template over the image and measuring the similarity. We provide a function for you to use to compute the output (called the *response image*) of this process. The mathematical details and full definition of the response image are presented in Section 6.1.2. Figure 1 provides an illustration of this process.

3. **Feature value.** Finally, we want to use this response image to compute a single feature value. Recall that we stored both the fragment and the location of its upper lefthand corner in the original image. Within the response image we will only consider a small neighborhood around this original location: in our implementation if the stored location was (x, y), we only consider the response values from positions $(x-3, y-3)$ to $(x+3, y+3)$. We take the maximum response value over this neighborhood, and this becomes our feature value. This is known as *max-pooling*.

To summarize, we are given an image and a feature dictionary consisting of fragments and locations. We take a fragment, "slide" it over the image and compute the similarity measure at each position. We then take the maximum similarity value over a small region around the

[2]See Antonio Torralba, K.P Murphy, and W.T. Freeman, "Sharing visual features for multiclass and multiview object detection." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 5, pp. 854-869, May, 2007



(a) (b) (c)

Figure 1: (a) A sample fragment, the one outlined in blue in the previous figure. (b) The new image we're considering. We "slide" the fragment over this image and at each location we compute a similarity measure between the fragment and the image. The value at position (x, y) of the response corresponds to the similarity measure between the fragment and the image region that is overlapped when the fragment is placed with its upper lefthand corner at position (x, y). Note that in order to keep the fragment within bounds its upper lefthand corner must stay within the region oulined in green. (c) The response image, which is exactly the size of the green region. White shows areas of high similarity.

original location of the fragment, and this is our feature value. We do this for all entries in the feature dictionary and obtain a feature vector corresponding to this image.

The sections below describe the math and the implementation details for each of these three steps.

6.1.1 Building a feature dictionary

For the milestone, we provide a dictionary for you, so you don't have to understand this section yet. If you choose to use these localized fragment-based features for the final submission, however, you will have to build your own dictionary. For the milestone dictionary we use just mugs, but for the final submission you can experiment with building individual dictionaries for each object or with using the same dictionary for all objects. Having separate features for each object may (or may not) lead to better classification accuracy; but sharing a dictionary may make your code run significantly faster.

To build a dictionary, we take the positive training examples resized to 32×32 pixels. We extract a set of fragments (image regions), ranging in size from 4×4 to 16×16, from random locations within the image, and store both the fragments and the original locations of their upper lefthand corner.

We extract 10 fragments from each image, bringing us up to 2990 entries in the dictionary. In order to keep the dictionary down to a manageable size and to find the best features to use for the milestone, we run a filtering step. The goal is to quickly find the most informative features and discard all others, in order to keep the running time of both the training and the testing phases down without (hopefully) sacrificing performance accuracy. We train a classifier using

boosted decision trees on a small subset of the training images (all 299 positives examples but only 2000 negative). We then discard all features that were not chosen by the decision trees, leaving us with just 119 entries in our dictionary. The assumption is that the features that remain are the most useful in discriminating between the classes in our training set, and can now be used to train a final classifier on the full training set in a reasonable amount of time.

There are many other equally valid approaches for choosing informative features, and you are free to experiment for the final submission.[3] You can also instead run an interest point detector[4] over the image to choose "interesting" locations from which to extract fragments. However, before you go ahead and implement any of these, you might want to try skipping the filtering step entirely and see if you can get satisfactory performance with just a random feature dictionary (the original Torralba et al. paper cited in section 6.1 did not do this kind of feature selection).

6.1.2 Computing the response image

To compute a response image we need to define a similarity measure between a fragment template T of size $w \times h$ and an image I of size $W \times H$. A standard technique is to use cross-correlation, where the similarity measure between the template and the image at position (x, y) is defined as

$$R_{\text{ccorr}}(x, y) = \sum_{x'=0}^{w-1} \sum_{y'=0}^{h-1} T(x', y') \times I(x + x', y + y') \tag{1}$$

A perfect match in this case is large, and a bad match will be small or 0. A more accurate similarity measure is normalized cross-correlation. Let

$$T'(x', y') = T(x', y') - \frac{1}{w \times h} \sum_{x''=0}^{w-1} \sum_{y''=0}^{h-1} T(x'', y'')$$

$$I'(x + x', y + y') = I(x + x', y + y') - \frac{1}{w \times h} \sum_{x''=0}^{w-1} \sum_{y''=0}^{h-1} I(x + x'', y + y'')$$

Then the normalized cross-correlation is defined as

$$R_{\text{nccorr}}(x, y) = \frac{\sum_{x',y'} T'(x', y') \times I'(x + x', y + y')}{\sqrt{\sum_{x',y'} T'(x', y')^2} \sqrt{\sum_{x',y'} I'(x', y')^2}} \tag{2}$$

The summations are again over $x' = 0, 1, \ldots w - 1$, $y' = 0, 1, \ldots h - 1$ (omitted to avoid clutter). The similarities in this case will be between -1 and 1 (perfect match)[5].

However, in practice normalized cross-correlation can be numerically unstable since there are regions of the image with very little variation, so $\sum_{x',y'} I'(x', y')^2$ in the denominator is small,

[3] For example, consider M.Vidal-Naquet and S. Ullman. "Object Recognition with Informative Features and Linear Classification." ICCV 2003.

[4] Consider: **1.** Mikolajczyk, K. and Schmid, C. "Scale & affine invariant interest point detectors." *International Journal on Computer Vision* 60(1):63-86, 2004; **2.** T. Kadir and M. Brady. "Scale, saliency and image description." *International Journal of Computer Vision*, 45(2):83-105, 2001; **3.** D. Lowe. "Distinctive image features from scale-invariant keypoints." *International Journal of Computer Vision*, 60(2):91-110, 2004

[5] The cross-correlation and normalized cross-correlation methods are both implemented in `openCV`'s `cvMatchTemplate` function, with method names `CV_TM_CCORR` and `CV_TM_CCOEFF_NORMED` respectively

or 0. To avoid this problem, the function that we provide for you is actually slightly different: we replace $\sum_{x',y'} I'(x',y')^2$ in the denominator with

$$\max\left(\sum_{x',y'} I'(x',y')^2, \quad \sum_{x',y'} T'(x',y')^2\right)$$

All of these similarity measures can only be computed for $x = 0, 1, \ldots W - w$, $y = 0, 1, \ldots H - h$ to avoid going out of bounds of the image I. Thus the response image will be of size $W - w + 1 \times H - h + 1$. This is important to understand since you will be allocating the response images yourself and working with them extensively to implement the milestone.

One final thing to be aware of is that the sums in the denominator can be computed very efficiently using *integral images* precomputed over the full image I. We will not go into detail here and we do not expect that you will want to change this, but if you do want to modify it, you can read more about integral images from various sources.[6]

6.1.3 Computing the feature value

To compute the feature value from the response image, in our implementation we use max-pooling over a 7×7 pixel region centered on the original location of the fragment. The dictionary we provide stores this region for you to use instead of just the original location. The advantage is that we have already taken care of ensuring that sliding the upper lefthand corner of the fragment template over this region will not cause it to go out of bounds of the image. If you choose to use this type of features for the final submission, you will have to build your own dictionary; be sure to consider this point.

6.2 Implementation

6.2.1 Loading the pre-selected features

We have provided you with a `FeatureDictionary` class that is capable of reading feature definitions from an XML file. We have also provided you with an initial set of feature defintions stored in `dict.xml` which are useful for detecting the mug class.

The `Classifier` class contains an instance of the `FeatureDictionary` class, called `_features`. This object will automatically load `dict.xml` for you. This means that for the milestone, you don't need to worry about discovering useful image features; they are given to you directly.

6.2.2 The template matcher class

We have also provided you with template matching code in `template.h` and `template.cpp`. The `TemplateMatcher` class can generate the template response images for you. All you need to implement in order to compute the features is the max-pooling operation described in section 6.1.3.

[6] E.g., Paul Viola and Michael Jones, "Rapid object detection using a boosted cascade of simple features," *International Journal of Computer Vision*, 2001

6.2.3 Sliding window object detection and image pyramiding

During training time, all of your images will be of the same size and you will only need to extract one vector of features from each image. During test time, in the `Classifier::run` method, you will need to search over a variety of locations and scales.

To do this efficiently, you will use a technique called image pyramiding. Your feature extractor will always operate on image windows of size 32×32 but you will vary the size of the input image in order to change the effective size of the feature extractor and recognize objects at multiple sclaes.

Specifically, you since you are guaranteed that all objects in the videos are of size at least 64×64 pixels, you should first resize the video frame to half its size to avoid wastefully analyzing regions that are too small. Run the object detector on resized image. Then you should shrink the image by a factor of 1.2, and run the object detector again. (Don't forget that you need to report detections in the coordinates of the original image, not the shrunken image). For best results, you should probably search over six different sizes of images, shrinking by a factor of 1.2 between successive layers.

For each size of image, you should evaluate the feature extractor and classify the feature vector at several different locations on a regular grid. This is called "sliding window object detection." We recommend using a grid spacing of 8 pixels by 8 pixels for the milestone, although if you implement non-maximum suppression for the final submission you might want to change it to, say, 4 by 4 pixels.

Note that the integral images mentioned in 6.1.2 are recalculated every time you call `TemplateMatcher::loadImage`. This means you should only call it once for each size of the video frame. Also, you should only need to call `TemplateMatcher::generateResponseImage` once for each feature and image size. You should not need to call it repeatedly to classify different locations within a scaled image. Calling any of the `TemplateMatcher` methods more often than necessary will make your classifier run extremely slowly.

7 Logistic regression

You should implement a logistic regression classifier that is trained during the execution of the `train` program, then re-loaded in the `Classifier::loadState` method during execution of the `test` program. Finally, you will use it to classify candidate regions of video frames in the `Classifier::run` method.

Logistic regression is described in Lecture Notes 5 on supervised learning. Be sure to add a bias term to the feature vector before inputting it into the logistic. You may also need to experiment with what learning rate to use for gradient descent.

You may find it useful to monitor learning by computing the mean squared error $||y - \hat{y}||_2^2$, where y is the true labels of the training data and $\hat{y}$ is the classifier's positive class probability estimate for each example. This gives you a measure of how close your predictions are to the actual labels that you may find easier to interpret than the log likelihood.

You have a variety of options for representing the vectors used in training the logistic model. We recommend using `std::vector`, but if you are not averse to learning a new library, you also have the option of using `Eigen`, which is a math library that allows the use of `Matlab`-like syntax in C++. We have provided a copy at `/afs/ir/class/cs221/stairvision/external/Eigen`.

8 The Competition

On the day of the competition, **Friday, December 14**, your object recognition program will compete against the programs of your classmates. The top three teams will receive some extra credit. We will not permit any "underhanded tricks" (such as labeling the objects yourself).

8.1 Structure

On the day of the competition, we will run your program on three new videos. As mentioned in section 5.2.1, only every 10th frame will be evaluated (starting with frame 5). Your program should return a list of `cObject`s for each evaluated frame of each video. We will then score the output to determine the winners.

8.2 Scoring

To enable you to score your classifier during development, we have provided you with an "evaluate" application which calculates a score given two xml files. One xml file should be your program's output (don't worry, test.cpp produces the xml for you; you only need to provide the rectangle coordinates and string labels as described previously) and the second xml file should be the ground truth labels that we provide you. To score your classifier on the video "easy", you would type:

```
 ./test -g /afs/ir/class/cs221/vision/data/easy.xml -o test.xml
        -x /afs/ir/class/cs221/vision/data/easy
./evaluate -m test.xml -g /afs/ir/class/cs221/vision/data/easy.xml
```

The actual score is calculated by first attempting to match each ground truth rectangle to one of the rectangles from your program's output. Two rectangles are considered to match if their string label is for the same type of object (e.g., both have the label "mug") and the area of their overlap is at least 60% as large as the area of the union of the two rectangles. Each ground truth rectangle can only be matched to one rectangle returned by your program, so you should avoid repeatedly labeling the same object.

If a rectangle returned by your program is matched to a ground truth rectangle, this is a *true positive*—your classifier got the correct result. If a rectangle returned by your program cannot be matched to a ground truth rectangle, this is a *false positive*—your classifier thought an object existed, but no object of that type is actually there. If a ground truth rectangle cannot be matched to a rectangle returned by your program, this is a *false negative*—your classifier thought no object of that type existed, when in fact one did.

Your classifier will be scored according to the F_1-*score*, which attempts to balance the need to have few false positives with the need to have few false negatives. The F_1-score is the harmonic mean of two other quantities: *precision* and *recall*. If we define tp as the nunber of true positives, fp as the number of false positives, and fn as the number of true negatives, then precision is defined as

$$p = \frac{tp}{tp + fp}$$

while recall is defined as

$$r = \frac{tp}{tp + fn}$$

More informally, precision is the proportion of labels produced by your classifier that are correct, while recall is the proportion of ground truth objects that are detected by your classifier.
Your overall score will be the harmonic mean of these two quantities, given by

$$F_1 = \frac{2 \times p \times r}{p + r}$$

The F_1 score ranges in value from 0 to 1. Your milestone project should get an F_1 score of about 0.25 for mugs on “easy”. For the final project, past students have scored as high as 0.9 on “easy”.
After you have finished your milestone project, you will probably notice that your classifier draws a very large number of rectangles at slightly different positions and of slightly different sizes around each object. Only one of these is counted as a correct label and the others are counted as false positives. Fixing this problem with a post-processing step is quite easy and will dramatically increase your score, so we recommend that you do so as soon as possible after the milestone.
Lastly, many classifiers have a sensitivity parameter. For the logistic regression model, it is the probability threshold above which we consider an object to be present. Tuning this parameter will determine whether your classifier tends to produce more false negatives or more false positives. Since the F_1 score attempts to balance the two kinds of errors, you can often improve your F_1 score greatly just by adjusting your threshold parameter, without changing the underlying learning algorithm or features. Feel free to talk to the TAs, particularly Ian, at office hours if you are concerned about whether your parameters are correctly tuned for the F_1 score to be a fair evaluation of your classifier’s performance.

8.3 Some more comments

1. The competition will be run on the `myth` machines; please make sure that your submitted code compiles and runs on these machines. Make sure that we will be able to compile your code by running `make`.

2. For the competition, we will require you to perform any training before the submission. Thus, the training time is not a constraint.

3. However, the program should be able to classify a 900 frame video within 2 hours. This is necessary so that the TAs can generate output for everyone’s programs before the competition begins. The version of the test program that the TAs will use to generate the output will automatically stop your program after 2 hours. If you are concerned that the algorithms you want to use will be too slow, be sure to ask the TAs how to use compiler optimization flags and how to optimize your code using `gprof` if you do not know how already.

4. We should be able to run your classifier by calling `test` using the `-c` flag to load a configuration file called `config.dat`. If you need to load more than one file, then make one file contain the paths to the others, e.g., “config.dat” could just contain the lines “foo.dat” and “bar.dat”.

5. If you have very large files that you cannot turn in using the submit script, then hardcode links to them in your own afs space. Just remember to use `fs acl` to give `rl` privileges to `system:anyuser` for the directory your files are in; otherwise we will be unable to access

them. Please do not put any symlinks in your submission directory; this causes trouble if we attempt to use cp -r on your submission directory. If you make symlinks to the cs221 class directory during development, please remove them temporarily before submitting.

6. Remember to release resources by calling the `cvReleaseImage(&IplImage)` method on any `IplImage` objects that are no longer used.
7. For accessing individual pixel values within `IplImage`s, be sure to use `OpenCV`'s `CV_IMAGE_ELEM` macro instead of `cvGet1D` or any other function; this will significantly speed up your code.
8. Please do not use any existing library implementations of logistic regression or fragment-based features for the milestone. However, after the milestone project you are encouraged to use OpenCV and other libraries. We particularly recommend the `CvBoost` class, which implements boosted decision trees.
9. Right now the only video available for you to test your project is `/afs/ir/class/cs221/vision/data/easy`, but we will release more as the quarter progresses.