The Intelligent Image

Max Jaderberg

Keble College, University of Oxford

Trinity Term 2012

Abstract

In this report, I shall discuss some awesome stuff.

Chapter 1

Introduction

The web contains billions of images, which are often the focus of attention on the web pages that include them. However, there is almost no information about the content of these images. Images on websites are purely binary data files, occasionally with some associated meta data included in the image's HTML[1] code. Very little information is available about the image, let alone the objects or scenes contained within the image. The aim of this project is to create a system which automatically recognises the objects within these images, thus releasing the information within them. This is a large scale object retrieval problem.

There are a large number of applications which would benefit from having detailed information on the contents of images. With more knowledge on the objects contained within images, one can create more effective search engines, better cataloguing and classification systems, user interfaces which engage viewers more, and relevant advertising based on the content. Novel applications could also be built, for example, which retrospectively embed geographical information in the image binary by recognising where the image was photographed. The plethora of useful applications provides a great deal of motivation for this project.

The result of the project is that a query image is inputted, the objects contained within the image are recognised, and the image is returned with the recognised objects "tagged" i.e. the region of the object in the image is outlined and it can be clicked to give relevant information. In this way the standard image has been transformed into the "intelligent image" - one which knows about the contained scene and can offer up information to the consumer about it's contents.

[1] Hyper Text Markup Language `http://en.wikipedia.org/wiki/HTML`

Figure 1.1: An example "Intelligent Image" showing the various hover over states

Recognising a wide range of objects in an image is not a trivial problem. For such a system to be useful the following needed to be addressed:

- Acquire and filter enough data to create a model to perform matching on a large range of objects.
- Create a retrieval system which provides accurate matching. False positive matching needs to be avoided as much as possible.
- Ensure matching can be done quickly on a database of millions of objects.

To recognise millions of different objects, reference images are needed to create a model to match against. From the outset, Wikipedia[2] is used as the primary model data source. Wikipedia is a crowd-sourced online encyclopaedia with many images contained within the articles, and is considered an accurate source of information. In the system, each Wikipedia page defines an object, with the images in the article being used to provide the data to match against. The content of the Wikipedia article is used to give the user further information about the object. A web crawler was written to extract the relevant images of desired Wikipedia pages and create the image database. Filtering is also performed to ensure only useful images are included in the database. The data sources are fully explained in Chapter 3.

Object retrieval uses a method employing a bag-of-words model. This builds upon the work described in x and y, more information of which is provided in Chapter 2. The visual words in the image database from Wikipedia are precomputed. At run time, the words in the query image are computed and searched against the database of precomputed words to find the top image matches. The top matches are spatially verified, with the first verified image being the result. Subsequent improvements to the base line system

[2] `http://en.wikipedia.org`

(described fully in Chapter 4) were made, including geometric improvements to the spatial verification and descriptors used to increase speed and accuracy of spatial verification, and matching improvements through query expansion using crowd-sourced data (dubbed "Turbo-boosting") from Microsoft's Bing[3] search engine. The geometric improvements and turbo-boosting is reported in Chapter 6 and Chapter 7 respectively.

Finally a website was developed to provide a front end interface for the system. A user can simply navigate to the website and upload an image. They then start the automatic tagging process, during which a realtime log of the process is displayed. After the query is complete, the intelligent image is displayed, which the user can interact with, see the names of the objects contained within the image, and click on the object to go to it's Wikipedia page. The software architecture of the website and the backend systems is explored in Chapter 5. The result is a realtime automatic tagging system which could recognise, for example, all the buildings in a tourist's photo album of London in seconds.

The aim of the project is to work towards recognising every object on Wikipedia, however due to time restraints a subset of objects was used for development and testing of the project. The subset chosen were the pages that appear on the Wikipedia page "List of Structures in London"[4].

[3] `http://www.bing.com`

[4] `http://en.wikipedia.org/wiki/List_of_structures_in_London`

Chapter 2

Background

Chapter 3

Data

There are three main datasets used by the application: the images from Wikipedia used to build the database of objects (Section 3.1), the images from Microsoft Bing used for the Turbo-boosting (Section 3.2) and the images from Google Images used for validation and testing (Section 3.3). All images that are used are resized so that their larger dimension does not exceed 1000 pixels to reduce storage space and provide homogeneity. This chapter describes the various datasets and how they are acquired. Table 3.1 gives an overview of the data used.

	Source	# Images	# Classes
Model images	Wikipedia	3963	732
Turbo-boosting images	Microsoft Bing	18273	732
Validation images	Google	701	294

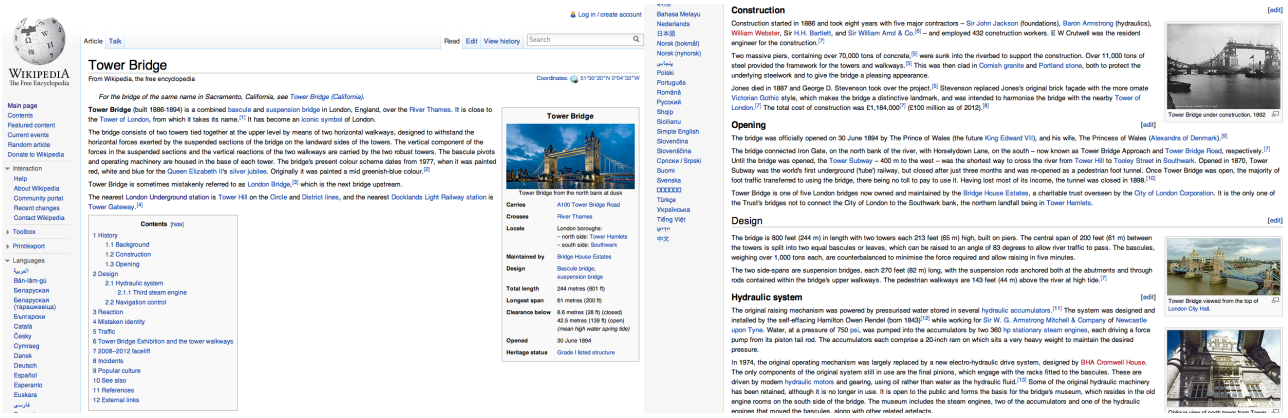
Table 3.1: A summary of the datasets.

3.1 Model Images

The model comprises of a dataset of images that depict the objects that are to be able to be recognised.

Each page of Wikipedia that contains images represents an object which can be matched. The database of images which is used to build the model is simply created by visiting each page on Wikipedia for the

objects desired and downloading the relevant images contained on the web page, labelling those images as being associated with the object. A script automates this process of building the model database.



(a) (b)

Figure 3.1: The Wikipedia page for "Tower Bridge". Note the images contained are those used in the model to represent this object.

To automate the downloading of object images from Wikipedia, Python[1] is used. WIkipedia offers a public application programming interface (API) over HTTP to access its data, as it is built on the MediaWiki framework[2]. However it is cumbersome and not easy to consume. Instead, a web crawler was written to explore Wikipedia pages and extract the relevant images.

A crawler object in Python finds all the images and notes the URLs of them for subsequent download. Firstly, the HTML of the Wikipedia page must be downloaded, as it appears to a web browser (an example of a web browser being Google Chrome). However, Wikipedia does not allow crawlers and automated bots to access its web pages. To overcome this, the HTTP header[3] of the crawler is edited to emulate that of a browser. This is implemented using the urllib2 library[4]. The code shown in Listing 3.1 shows an example of how to read the HTML of the main Wikipedia homepage. The HTML document for each Wikipedia page is parsed using the BeautifulSoup library[5]. All the anchor elements are found and stored for further

[1] `http://www.python.org`

[2] The MediaWiki framework was originally developed for Wikipedia and provides an API over HTTP as standard. `http://www.mediawiki.org/wiki/API` provides documentation for the API.

[3] `http://www.w3.org/Protocols/rfc2616/rfc2616.html` describes the Hyper Text Transport Protocol and the various header fields.

[4] `http://docs.python.org/library/urllib2.html`

[5] `http://www.crummy.com/software/BeautifulSoup/`

crawling. The images contained within the HTML are also found by looking within the part of the HTML document that is unique to the specific Wikipedia page (see Listing 3.2).

Listing 3.1: The code used to gain access to Wikipedia's content using a crawler by emulating a browser.

```
## Python
import urllib2
# Emulate the user agent as that of a browser
user_agent = "Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.7)
    Gecko/2007091417 Firefox/2.0.0.7"
headers = {"User-Agent": user_agent}
# Request the webpage
req = urllib2.Request("http://en.wikipedia.org", headers=headers)
resp=urllib2.urlopen(req)
# Read the HTML of the response
html = resp.read()
```

Listing 3.2: Parsing the Wikipedia article HTML document to find the relevent images.

```
## Python
def _get_content_body(self, soup):
        main_content = soup.find('div', {"class": "mw-content-ltr"})
        if main_content is None:
            return None
        # remove navboxes
        navboxes = main_content.findAll('table', {'class': 'navbox'})
        [navbox.extract() for navbox in navboxes]
        return main_content

def _get_image_links(self, soup):
        return soup.findAll('a', {'class': 'image'})

# Parse the HTML
soup = BeautifulSoup(html)
# Get the main article body
soup = _get_content_body(soup)
# Get a list of image links
image_links = _get_image_links(soup)
```

The output of the crawler is a CSV file of the image URLs and the object class the images belong to. The object class is simply named from the URL of the Wikipedia page (for example all images appearing on `http://en.wikipedia.org/wiki/Tower_Bridge` will have class `Tower_Bridge`).

The images mentioned in the CSV file produced by the crawler are then downloaded to local storage. Each image that appears on Wikipedia has a "file page" which displays the image along with properties and metadata on the image[6]. This "file page" is visited for each image, and the page is parsed to extract the storage URL of the image as well as its file format and original size. As the application resizes all images that exceed 1000 pixels to 1000 pixels, it is a waste of time and storage space to download the original

[6]For an example see `http://en.wikipedia.org/wiki/File:Tower_bridge_London_Twilight_-_November_2006.jpg`

image and later resize it. Instead, Wikipedia's inbuilt thumbnail engine is exploited, which resizes the image on Wikipedia's servers and allows you to download a thumbnail of a user selected width[7]. Therefore if the original image on Wikipedia exceeds 1000 pixels, the 1000 pixel thumbnail version is downloaded instead. The images downloaded are saved in a folder named after it's class. The result is a directory containing a folder for each class, within which are the images for that class.

This process creates a structured dataset of model images from the Wikipedia pages visited. For the List of Structures in London dataset used, there were 732 classes which had 3963 images associated with them (on average 6 images per class).

3.2 Turbo-boosting Images

To achieve effective turbo-boosting, many additional images are needed to supplement the model images acquired from Wikipedia. Microsoft Bing is used as the source of the turbo-boosting images. For each class, 25 additional images are downloaded to boost that class.

Bing offers a public API that can be used to perform image searches programmatically. After obtaining an application ID from Bing for authentication, complex search requests can be made over HTTP, with the results returned in JavaScript Object Notation (JSON) format[8].

Bing image search takes a number of keywords - the query - and returns a list of images from web pages related to the query. Further filters can be applied to further narrow down the search to the most relevant images. This is shown in Figure 3.2 on the website version of Bing. The "Style" and "Size" filters are especially useful in this application, as all images should be photographs for the List of Structures in London dataset, and large images are preferable so as to include as much detail as possible. Setting these filters precludes many instances of graphics and logos which are not suitable for turbo-boosting.

All the parameters that appear in the web interface for Bing image search can be replicated in the API request with query parameters. A MATLAB script is used to consume the API and download the images. Listing 3.3 shows the URL used for the API call to get the search results for the images for a particular

[7] `http://www.algorithm.co.il/blogs/programming/wikipedia-images/` describes how this is exploited

[8] JSON is an alternative to XML for representing structured data. `http://www.json.org/` provides more information.

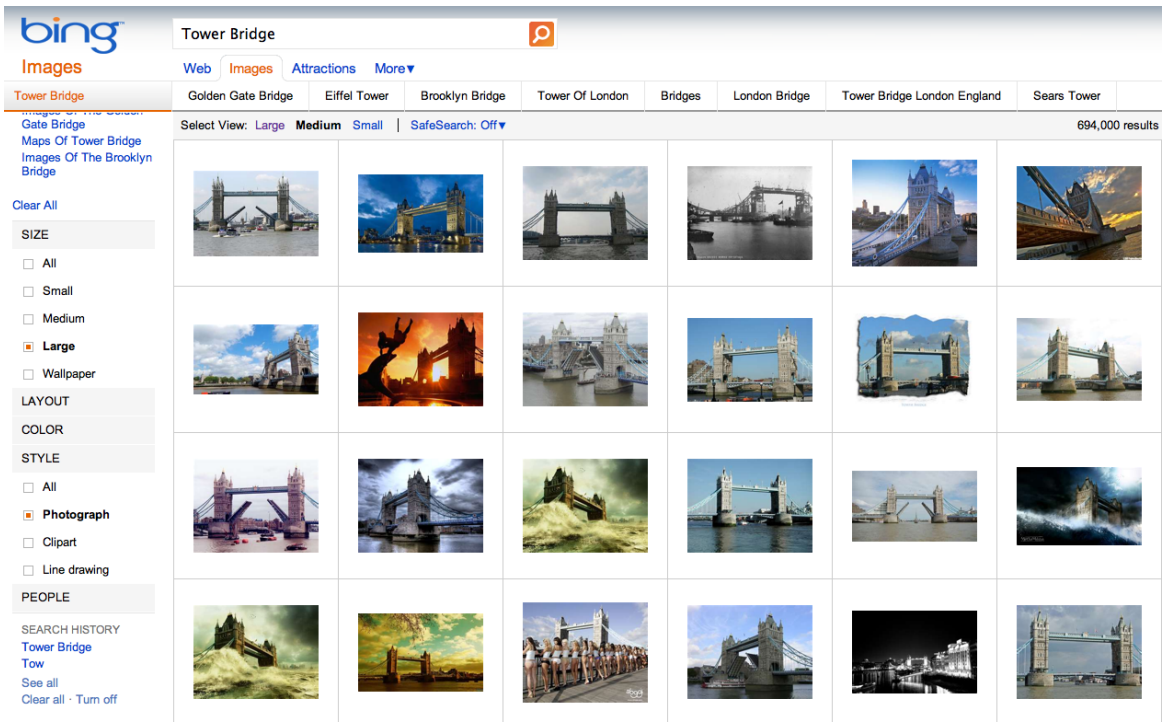


Figure 3.2: The browser interface of Bing image search which is replicated in the API query URL. Note the filters in the left column.

class. The URL encoded[9] class name is used as the query. For example, for the class `Tower_Bridge`, the variable `search_term` appearing in Listing 3.3 will be set to `"Tower%20Bridge"` (note `" "` is URL encoded as `"%20"`). Both the "Style" and "Size" image filters are set to "Photo" and "Large" respectively by setting the `Image.Filters` parameter.

Listing 3.3: A Bing image search API request.

```
%% MATLAB
request_url = ['http://api.bing.net/json.aspx?' ...
                 'AppId=' app_id ...
                 '&Query=' search_term ...
                 '&Sources=Image' ...
                 '&Version=2.0' ...
                 '&Adult=Strict' ...
                 '&Image.Count=' nPhotos ...
                 '&Image.Filters=Style:Photo+Size:Large' ...
                 '&JsonType=raw' ...
               ];
% Read the result of the request
response = urlread(request_url);
% Parse the result from JSON to MATLAB structure form
resp_struct = parse_json(response);
```

[9]URL encoding ensures that all characters are in a form which can be used as a URL. See `http://www.w3schools.com/tags/ref_urlencode.asp` for more information.

Using MATLAB's inbuilt `urlread` function, the search results are requested and returned in JSON format. The JSON result is then parsed and converted into a MATLAB structure object for reading. An example JSON response is shown in Listing 3.4. Each element in the array `Results` is an image result. Each image is then downloaded from its `MediaUrl` field using a modified version of the `imread` function which allows for request timeouts, as some image resources may have expired since their submission to the Bing database.

Listing 3.4: A Bing image search response in JSON format. The "Results" array is truncated to one element.

```
// JSON
{
   "SearchResponse":{
      "Version":"2.0",
      "Query":{
         "SearchTerms":"Tower Bridge"
      },
      "Image":{
         "Total":780000,
         "Offset":0,
         "Results":[
            {
               "Title":"Tower Bridge - London Photo (551176) - Fanpop",
               "MediaUrl":"http://images.fanpop.com/images/image_uploads/Tower-Bridge.jpg",
               "Url":"http://www.fanpop.com/spots/london/images/551176/title/tower-bridge",
               "DisplayUrl":"http://www.fanpop.com/spots/london/images/title/tower-bridge",
               "Width":1600,
               "Height":1200,
               "FileSize":761104,
               "Thumbnail":{
                  "Url":"http://ts3.mm.bing.net/images/thumbnail.aspx?q=4757...",
                  "ContentType":"image/jpeg",
                  "Width":160,
                  "Height":120,
                  "FileSize":3274
               }
            }, ...
         ]
      }
   }
}
```

The downloaded images are resized if larger than 1000 pixels and stored in a folder named after its class. As for the model images, the result is a directory containing a folder for each class, within which are the turbo-boosting images for that class.

3.3 Validation Images

Images are needed to test and validate the yield performance of the object recognition system. Therefore a separate dataset of images with their ground truth classes is needed. Ideally all classes would be tested and each test image should be a fair representation of the object of that class. Google image search was used first to automatically download 8 images for each class. The images were then checked manually to refine the dataset.

Google image search is very similar to Bing image search described in the previous section. However the results are markedly different, providing another set of images that are perfect for testing. Google offers an API over HTTP which can be used to search based on a text query and, as with Bing, filters can be applied. The result is returned in JSON format.

Listing 3.5: A Google image search API request.

```
%% MATLAB
request_url = ['https://ajax.googleapis.com/ajax/services/search/images?v=1.0' ...
        '&q=' search_term ...
        '&as_filetype=jpg' ...
        '&imgsz=xxlarge' ...
        '&imgtype=photo' ...
        '&rsz=8' ...
];
% Read the result of the request
response = urlread(request_url);
% Parse the result from JSON to MATLAB structure form
resp_struct = parse_json(response);
```

Listing 3.5 shows the formulation and request of a Google search API request. As with the Bing requests, the search term is the URL encoded class name. The JSON response (Listing 3.6) is parsed into a MATLAB object and the `url` field is used to download the image.

Listing 3.6: A Google image search response in JSON format. The "results" array is truncated to one element.

```
// JSON
{
   "responseData":{
      "results":[
         {
            "GsearchResultClass":"GimageSearch",
            "width":"1024",
            "height":"819",
            "imageId":"ANd9GcSTU9Qv93OE3Q5kjo0h5Dcsl6sYBGURxSlmw8tfUbISx6heecXcY9VKkquZ",
            "tbWidth":"150",
            "tbHeight":"120",
            "unescapedUrl":"http://www2.hiren.info/desktopwallpapers/natural/the-tower-...",
            "url":"http://www2.hiren.info/desktopwallpapers/natural/the-tower-bridge.jpg",
            "visibleUrl":"www.hiren.info",
```

```
            "title":"Desktop Wallpapers  Natural Backgrounds  The \u003cb\u003eTower
                Bridge\u003c/b\u003e \u003cb\u003e...\u003c/b\u003e",
            "titleNoFormatting":"Desktop Wallpapers  Natural Backgrounds  The Tower Bridge
                ...",
            "originalContextUrl":"http://www.hiren.info/desktop-wallpapers/natural-pic...",
            "content":"The \u003cb\u003eTower Bridge\u003c/b\u003e, London,",
            "contentNoFormatting":"The Tower Bridge, London,",
            "tbUrl":"http://t1.gstatic.com/images?q\u003dtbn:ANd9GcSTU9Qv93OE3Q5kjo..."
         }, ...
      ], ...
   "responseDetails":null,
   "responseStatus":200
}
```


As with the turbo-boosting images, the downloaded images are resized if larger than 1000 pixels stored in a folder named after its class. Again, the result is a directory containing a folder for each class, within which are the turbo-boosting images for that class.

After automatic download of potential test images for each class, the dataset was checked over manually. Images that appear in the model dataset, as well as images that do not fairly depict the class they are to test are removed from the test set .

Chapter 4

Baseline system

The base line system draws upon the research and literature mentioned in Chapter 2. Upon starting the project, an application was provided that includes a basic database creation and image matching process based on a dataset of structures in Oxford. This application was expanded upon to create the current system.

There are two separate processes forming the project. The first is the pre-computation process that creates the databases and data structures required for object recognition. The second is the object recognition process that takes a query image and returns the names and locations of the objects in the image. A summary of these two processes is described in Figure 4.1 and Figure 4.2 respectively.

The pre-computation process is run once on the dataset to produce the database and working data structures required for the object recognition process. For each model image, the features are detected and associated descriptors generated. This is described further in Section 4.1. A sample of the features are then used to generate the visual word vocabulary (Section 4.2). The histograms of visual words are then weighted and collected into an index ready for querying (Section 4.3). This completes the basic pre-computation process and the system is ready for use.

The object recognition process takes a query image and attempts to recognise the objects contained within the image. Firstly, the feature descriptors are computed for the detected features within the query image. The visual words are computed based on the vocabulary created during the pre-computation process, and the weighted histogram produced for the query image. A search is then performed on the index of

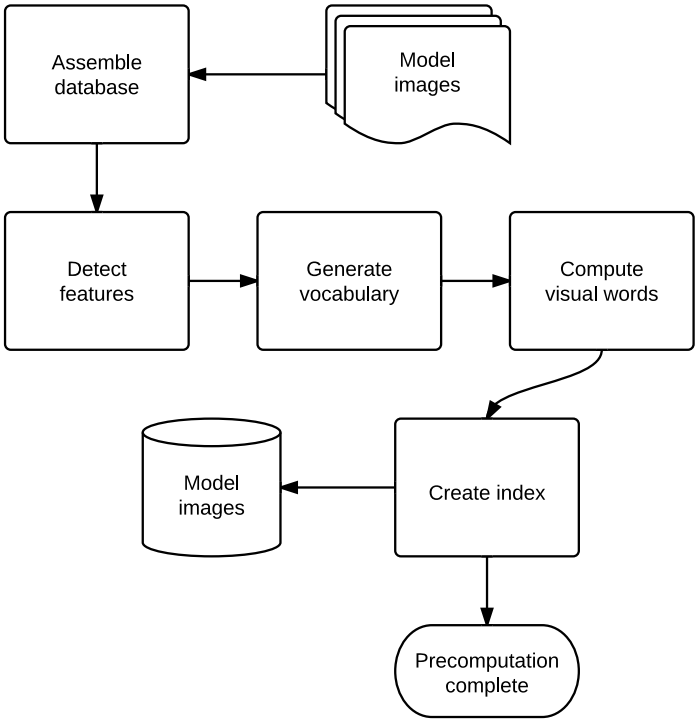


Figure 4.1: The flow diagram for the basic pre-computation process.

histograms for the model images (see Section 4.3) the output of which is a list of images based on how highly they match the query image. Going down the list of top matches by histogram, spatial verification is performed to ensure the visual words in both the query image and match image form the same shaped object (all objects are assumed rigid). This is described in Section 4.4. Once a match has been spatially verified, this object is deemed recognised and added to the list of found objects. Multiple matching is then performed by repeating this process, excluding regions of the image containing a previously recognised object (Section 4.5).

The remainder of this chapter describes further details of the parts of the processes described above.

4.1 Feature Detection and Description

The feature detection and description methods used are the original scale-invariant feature transform (SIFT) algorithms. The advantages of using SIFT are that the detected features and their descriptors are invariant to image translation, scaling, and rotation, partially invariant to illumination changes and robust to local geometric distortion. This is essential to be able to match the same object features across varied sources

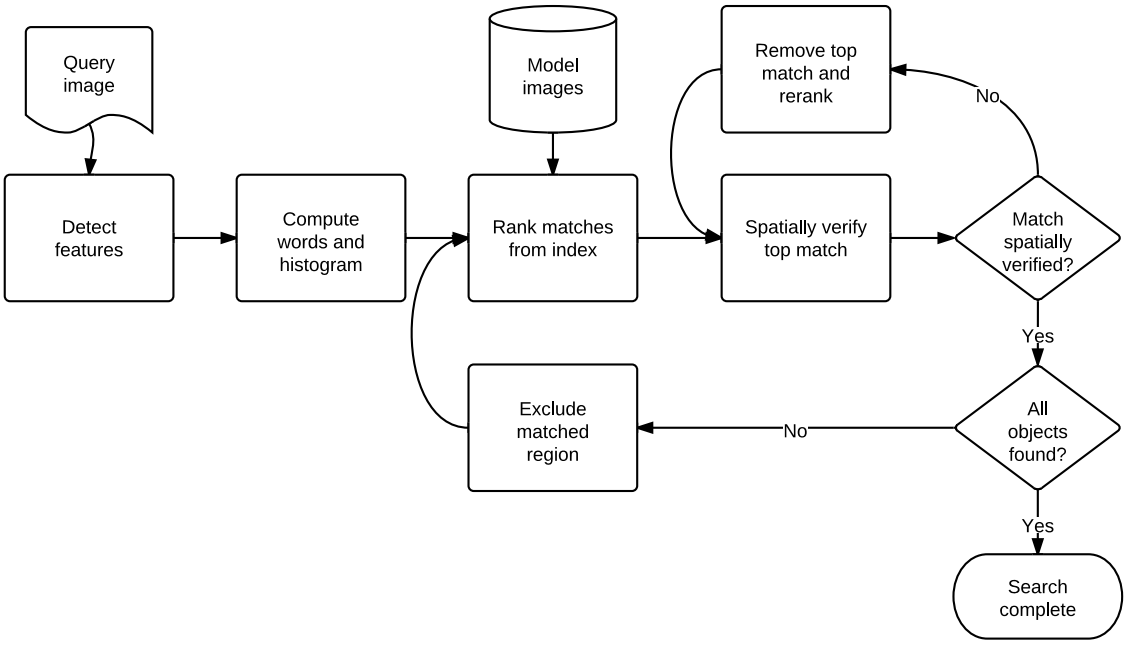


Figure 4.2: The flow diagram for the basic object recognition process.

of images.

The SIFT features are defined as maxima and minima of the result of difference of Gaussians function[1] applied in scale-space to a series of smoothed and resampled images. Low contrast candidate points and edge response points along an edge are discarded. These features are then described by the SIFT feature descriptor - a 128-dimensional vector.

The result of the SIFT feature detection and description algorithm are two matrices. The first is a matrix of feature points or frames, Equation 4.1, where each column describes the position (x_i, y_i) in the image, the scale s_i and orientation θ_i for feature i. Corresponding to the frames matrix is a descriptor matrix, Equation 4.2, where each column is the 128-D vector $\mathbf{d_i}$ that describes feature i.

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_{N-1} & x_N \\ y_1 & y_2 & \cdots & y_{N-1} & y_N \\ s_1 & s_2 & \cdots & s_{N-1} & s_N \\ \theta_1 & \theta_2 & \cdots & \theta_{N-1} & \theta_N \end{bmatrix} \tag{4.1}$$

$$\begin{bmatrix} \mathbf{d_1} & \mathbf{d_2} & \cdots & \mathbf{d_{N-1}} & \mathbf{d_N} \end{bmatrix} \tag{4.2}$$

[1] http://en.wikipedia.org/wiki/Difference_of_Gaussians

4.2 Visual Words

To avoid matching features in unbounded, 128-dimensional space, the SIFT features are quantised. These quantised SIFT features are known as visual words.

During the pre-computation process, the vocabulary of words is created. The vocabulary is essentially the clustering of SIFT space. The number of clusters (visual words) to be created is the vocabulary size, 100,000 for this application. Vocabulary creation is done using the approximate nearest neighbours K-means algorithm. The clustering is performed on a random sample from all the feature descriptors for the entire model images dataset. The number of features sampled was 30 times the size of the vocabulary, i.e. 3 million.

The result of the vocabulary creation is a kd-tree which can be used to get the word associated with a SIFT descriptor. Each word is assigned an ID, and the images can then be represented as a list of words, where each word is the nearest visual word to the SIFT descriptor.

4.3 Histograms and Index

The images are represented by a list of words as described in the previous section. This list of words can in turn be represented as a histogram, with each element containing the number of occurrences in the image of the word with ID equal to the element number. Therefore each histogram is a sparse 100,000 element array. Two examples of the raw histograms are shown in Figure 4.3c and Figure 4.3d.

The term frequency-inverse document frequency[2] (tf-idf) weights are then computed for each word across the entire dataset of model images. These weights are applied to the histograms to down-weight common, uninformative visual words and up-weight unique, informative visual words. The differences in histograms before and after weighting are illustrated in Figure 4.3.

Each model image therefore has a histogram that is used for matching. All the histograms are packaged into a matrix (each column being an image's histogram) that is used as the index for query matching.

To find the most similar images to a query image, a dot product is performed between each of the

[2]This is a practice developed originally for text search. See `http://en.wikipedia.org/wiki/Tf*idf` for more information.

model image histograms and the query image histogram. This is shown in Equation 4.3, where $\mathbf{h_{query}}$ is the tf-idf weighted histogram of the query image and $\mathbf{h_i}$ is the tf-idf weighted histogram of model image i. $\mathbf{scores}$ is a $N \times 1$ array with the result of the dot product with each model image's histogram in each element. The matches are ranked in decreasing order of their score - the model image with the highest score is said to be the most similar to the query image.

$$\mathbf{scores} = \mathbf{h_{query}^T} \begin{bmatrix} \mathbf{h_1} & \mathbf{h_2} & \cdots & \mathbf{h_{N-1}} & \mathbf{h_N} \end{bmatrix} \tag{4.3}$$

4.4 Spatial Verification

Matching based purely on the tf-idf weighted histograms is effective, however it is prone to false positives. This is due to the fact that two different objects may have very similar features (and therefore many similar visual words), but these features are in very different places as they are not the same object. Therefore, a final spatial verification must be done to ensure that the visual words that appear both in the query image and the model image form the same rigid bodied object.

The spatial verification restraint in this application is that there should exist an affine transformation between the scene in the query image and the scene in the model image that is the potential match. Practically, this means that there should be an affine transformation that maps the visual words in the query image to the position of the same visual words in the model image. Equation 4.4 shows this transformation, such that $\mathbf{x}'$ is the positions of the visual word in the model image, $\mathbf{x}$ is the position of the visual word in the query image, and $\mathbf{H}$ is the affine transformation between the two images.

$$\mathbf{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{Hx} \tag{4.4}$$

The random sample consensus algorithm[3] (RANSAC) is used to estimate an affine transformation between the two images, using the corresponding visual words in each image. The result of the RANSAC

[3] http://en.wikipedia.org/wiki/RANSAC

spatial verification is the number of visual words in the query image that map to the correct positions in the model image (within some tolerance region). Matches that have enough inliers under the transformation are said to be spatially verified - that is the estimated affine transformation is accurate and both images depict the same rigid bodied object.

Figure 4.4 shows the result of spatial verification on matched SIFT features. Note the disregarding of some matches after RANSAC as these matches do not conform with the estimated affine transformation.

Spatial verification is performed on each model image in descending order of the tf-idf histogram matching score. The first model image to be successfully spatially verified is deemed to be an accurate match and the process is terminated, with the class the model image represents being the object found. The region of the query image that is labelled as the object is the bounding box of visual words that spatially match the model image.

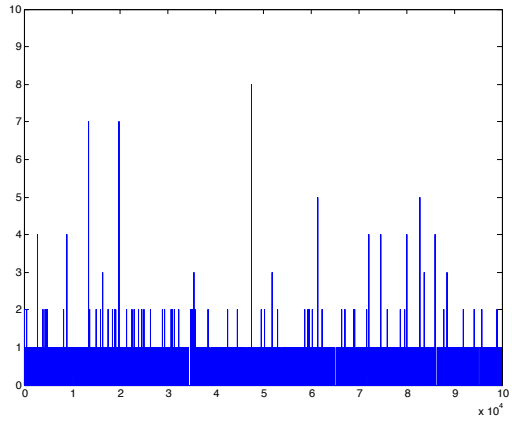
4.5 Multiple Object Matching

The object recognition engine can recognise multiple objects in a single image (an example is shown in Figure 4.5). Once an object has been successfully recognised, the query is re-issued with the same query image, however the visual words contained within the regions of already recognised objects are excluded from the query process. Previously recognised objects are ignored as matches from the re-issued queries and the process finishes when no new objects can be found.

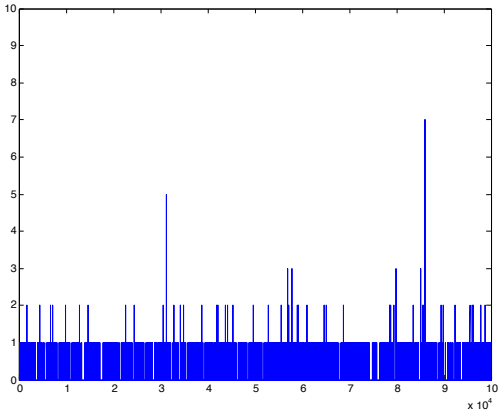
This method allows any number of objects to be recognised within a single image. The downside of this process is that it requires multiple queries so increases the time taken to complete.

(a) Tower Bridge image 1

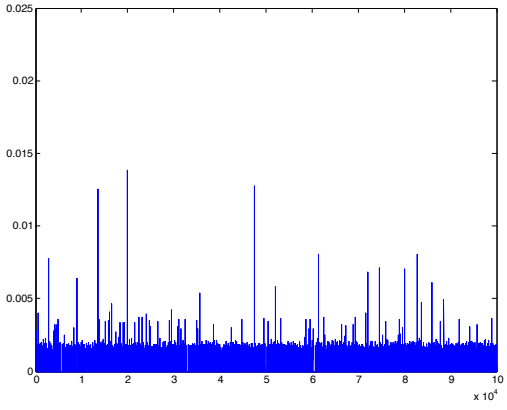
(b) Tower Bridge image 2



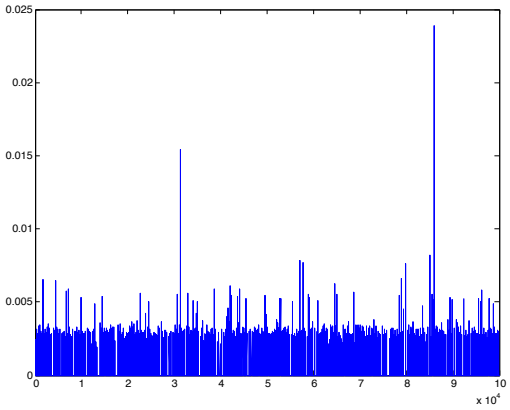
(c) Raw histogram for image 1



(d) Raw histogram for image 2



(e) Weighted histogram for image 1

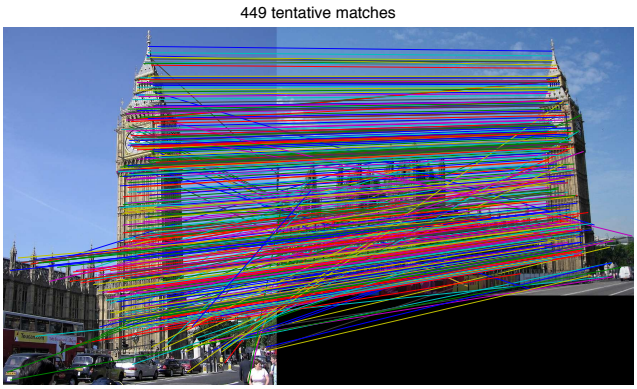


(f) Weighted histogram for image 2

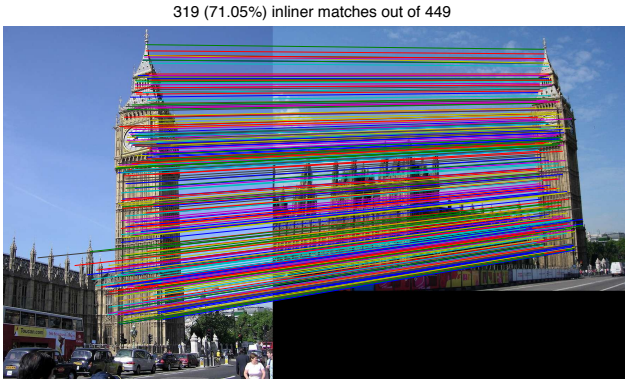
Figure 4.3: Two images for the class `Tower_Bridge` with their raw histograms and tf-idf weighted histograms

(a) Big Ben image 1

(b) Big Ben image 2



(c) Matches based purely on SIFT feature similarity



(d) Matches after RANSAC spatial verification

Figure 4.4: RANSAC spatial verification performed on two images of Big Ben.

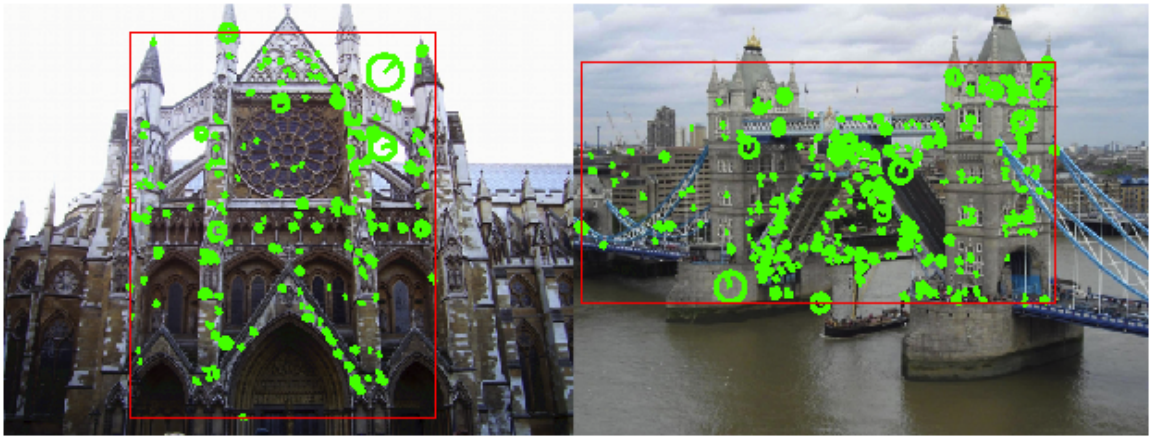


Figure 4.5: Multiple matched objects in a single query image. Note: this example query image has been artificially created to include two objects.

Chapter 5

Implementation

While Chapter 4 outlines the processes involved with the object recognition engine, this chapter describes the practical implementation of the system including the front end web application for consumer interaction.

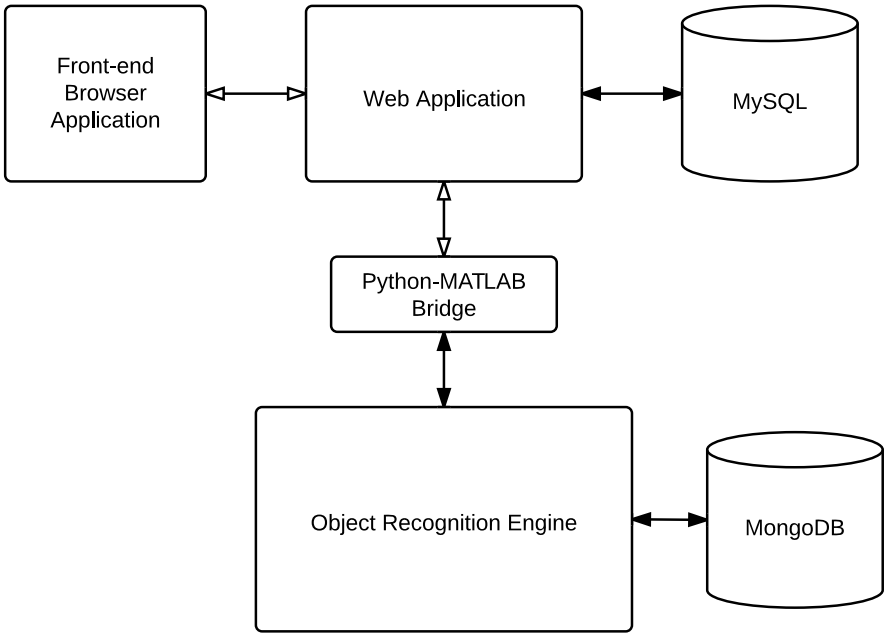


Figure 5.1: An overview of the system architecture.

Figure 5.1 gives a holistic view of the system architecture. The solution is divided in to two separate parts: the object recognition engine and the web application. These are described in Section 5.1 and Section 5.2 respectively. These two parts are completely independent modules by design, however they are connected via a Python-MATLAB bridge, the details of which are given in Section 5.3.

The resulting product is a web site with an intuitive user interface. A user can upload a photo, and the photo will be presented back to the user with the objects tagged.

5.1 Object Recognition Engine

The object recognition engine performs the pre-computation and recognition processes described in Chapter 4. This is all implemented using MATLAB. MATLAB was chosen due to its ease in image manipulation, rich toolboxes, and the provided starter application was written in MATLAB.

The resulting interface is a MATLAB function `get_objects()` that takes a single argument containing the location of the query image to recognise the objects in. The output is a structure of the object classes found and their bounding rectangles in the query image. The function takes around 5-10 seconds to run.

5.1.1 Database

A filesystem is used for the storage of model images (see Chapter 3), however a working record is needed of the dataset for book keeping and quick searching. For this, MongoDB[1] is used.

MongoDB is a NoSQL[2] database application that stores data in the form of documents - data structures similar to JSON objects. The data is unstructured, and provides a simple way of storing data without dealing with the complexities of tables and relationships that come with relational databases (e.g. SQL). Listing 5.1 shows a model image stored in the MongoDB database. The fields of the documents (i.e. `"_id"`, `"name"`, `"path"`, `"class"`, `"size"`) are fully searchable and can be indexed for faster lookup[3].

Listing 5.1: A document representing a model image stored in MongoDB.

```
// BSON
{
        "_id" : ObjectId("4f7317501b515eaa6f148c5d"),
        "name" : "1222|10_Downing_Street.jpg",
        "path" : "~/4YP/data/d_ransac/images/10_Downing_Street/1222|10_Downing_Street.jpg",
        "class" : "10_Downing_Street",
        "size" : {
                "width" : 800,
                "height" : 457
        }
```

[1] `http://www.mongodb.org`

[2] NoSQL is a family of databases that does not require fixed table schemas, and are sometimes known as "structured storage".

[3] By deafault, the `"_id"` field is indexed for fast lookup by ID.

```
}
```

MongoDB is run as a daemon and the directory of storage files is set to be a directory within the solution's working directory. This allows complete segmentation of each database. The daemon sets up a web server on port 3036 which can accept any number of connections over TCP to access the database.

The interface with MATLAB is done using the Java library provided by MongoDB[4]. MATLAB has the ability to run Java inside it, so the Java MongoDB library can be consumed in a fairly natural way.

The frames, descriptors, words and histograms for the model images are stored on disk as MATLAB binary files. They are associated to the database entries via their filename which includes the ID assigned by MongoDB to the image's document. For example, for the model image `1222|10_Downing_Street.jpg` shown in Listing 5.1 the frames file will be stored in the frames directory with the filename `4f7317501b515eaa6f148c5d-frames.mat`.

5.1.2 Image Processing

The bulk of the image processing and processor intensive operations are outsourced to the VLFeat library[5]. VLFeat is an open source library implementing various computer vision algorithms in C. Being implemented in C means that the algorithms compute faster and more efficiently than if they were written in MATLAB. The VLFeat provides MATLAB wrappers to the functions which are use to interface with the MATLAB engine.

To enable VLFeat, `run vlfeat/toolbox/vl_setup` is executed on startup, after which all the VLFeat functions can be used. `vl_sift()` is used for SIFT feature detection and descriptor computation, and `vl_kdtreebuild()` and `vl_kdtreequery()` are used for efficient visual word creation and retrieval.

5.1.3 Distributed Pre-computation

The pre-computation process takes a long time to complete, as every image of the thousands in the dataset must be processed and the clustering algorithm must be run. However, the majority of the functions within

[4] See `http://www.mongodb.org/display/DOCS/Java+Tutorial` for documentation.

[5] `http://www.vlfeat.org`

the pre-computation involve independent computations on each image in turn. This means that these steps can be run in parallel.

A room of computers in the Engineering Science Department is used for the parallel pre-computation. These Linux machines are accessed via SSH[6]. In total there are around 30 machines that are used.

The distributed pre-computation is conducted by a Python script using the Fabric library[7] for simplifying the SSH tasks. The current build is zipped up locally and deployed to the remote shared disk. The Python script then distributes the jobs across all the hosts that can be connected to. All jobs generate a finished flag so the pre-computation script can keep track of the dispatched jobs and synchronise the process.

Listing 5.2: The bash script used to start MATLAB.

```
#!/bin/bash
unset DISPLAY
nohup matlab -nodesktop -nosplash -nodisplay -r "try,$1($2,$3,'$4','$5'),catch err,
    f=fopen('error_logs/$2-$1-error.txt','w'); fprintf(f,err.message); fclose(f); exit,
    end, exit" -logfile matlab_logs/matlab_log$2.txt > nohup_logs/nohup$2.out 2>&1 &

# Example usage for machine 11 out of 28:
sh dist_matlab_suppress.sh dist_compute_features 11 28 engs-station41.eng.ox.ac.uk
    engs-station51.eng.ox.ac.uk
```

Listing 5.3 shows the script used to launch a MATLAB job. The MATLAB function must take four arguments: `n_split`, `N_split`, `first_host`, `this_host`. These describe the current machine's number, the total number of machines, the address of the first machine and the address of the current machine. These are required for the splitting of the data and to connect to the MongoDB server which is always run on the first machine.

Using the distributed system for pre-computation and validation [WHERE DO I WRITE ABOUT THE VALIDATION AND RESULTS?!] results in a far quicker process, reducing the time from 15-20 hours for some jobs to around 1 hour.

5.2 Web Application

The web application sits in front of the object recognition engine and implements a practical application of the engine. The function is a website that allows users to upload photos for automatic tagging of objects.

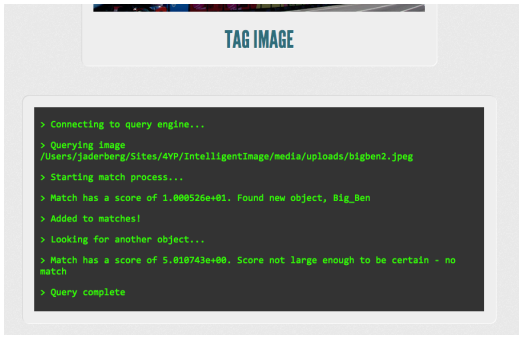
[6] http://en.wikipedia.org/wiki/Secure_Shell

[7] http://docs.fabfile.org/en/1.4.0/index.html

The web application is divided in to two parts, the front-end application that is runs in a users web browser (Section 5.2.1) and the back-end web server (Section 5.2.2) that handles the requests and responses from the front-end.

(a) Stage 1 - initial view

(b) Stage 2 - start matching process



(c) Stage 3 - log of matching process

(d) Stage 4 - the intelligent image

Figure 5.2: Screenshots showcasing the web application's functionality.

Figure 5.2 shows the steps for a user using the website which are as follows:

1. Select and upload an image.
2. Click "Tag Image" to start the matching process.
3. View the log of the matching process.
4. When match is complete, view and interact with the "intelligent image".

5.2.1 Front-end

The front-end is a web page viewed in a web browser. It consists of three components: the HTML code that defines the structure and layout of the web site, the CSS[8] that defines the styling and the Javascript that provides the logic for the website.

The Javascript scripts make heavy use of the JQuery library[9]. JQuery abstracts and simplifies manipulation of HTML elements on the page as well as providing built in animation and HTTP request functions.

The goal of the website is to appear smooth and responsive, therefore all requests after the initial loading of the page are done asynchronously. This means that Javascript handles all subsequent HTTP requests and responses in the background, without refreshing the browser state. This is done using JQuery's `getJSON()` function, where the response data is packaged in JSON format.

The website consists of the four stages described above, and there is an HTML `div` (a container element) for each stage. The CSS styles these to look indented in the page to provide a point of focus for the user. The four containers for each stage are all included in the initial HTML. However, only the stage that is currently active is shown by adding the following attribute to the other containers: `style="display: none;"`.

Initially, the user is presented with an image upload form (see Figure 5.2a). Once an image has been selected, a Javascript event is fired, prompting the automatic asynchronous upload of the image. The server responds with HTML to display the image which is inserted in the next container and shown (Figure 5.2b).

Listing 5.3: The HTML code first stage container which includes the image upload form.

```
<!--HTML-->
<div id="upload-box" class="smooth-box">
        <div style="height: 70px;">
                <img id="upload-img" src="/img/down_arrow.png">
                <h1>Upload an image...</h1>
        </div>
        <div class="clearit"></div>
        <div class="form-holder">
                <form method="post" action="/upload" enctype="multipart/form-data">
                    <input type="file" name="image">
                    <input type="submit" style="display: none;">
                </form>
        </div>
```

[8] Cascading style sheers - more information at `http://www.w3schools.com/css/`

[9] `http://jquery.com`

```
</div>
```

Upon clicking the "Tag Image" button, the real-time log is displaying informing the user of the progress of the matching process (Figure 5.2c). The real-time log is achieved by the MATLAB process writing a log file and the Javascript app periodically asking the web server for the contents of the latest version of that file.

Once the match process is complete, the "intelligent image" HTML (shown in Listing 5.4) is returned to the Javascript app. The image is shown with the tags for the objects overlaid. On hover over, the border of the tag becomes solid and the name of the object is shown (Figure 5.2d). On click, the Wikipedia page for the object is opened. This is achieved purely with HTML and CSS.

Listing 5.4: The HTML code for the intelligent image.

```
<!--HTML-->
<div class="tagged-img-wrap">
       <img id="tagged-img" src="/uploads/query_image.jpg}">
       <a style="left: 105px; bottom: 232px;" class="img-tag"
           href="http://en.wikipedia.org/wiki/Tower_Bridge}" target="_blank">
              <div style="width: 303px; height:288px;" class="tag-border"></div>
              <span class="tag-label">Tower Bridge</span>
       </a>
</div>
```

5.2.2 Back-end

The back-end web server is in place to handle all incoming HTTP requests and produce appropriate responses.

In this application, a Python web server using the Django framework[10] is used. The HTTP requests are routed to different Python functions depending on what URL is requested, and these functions handle the processing of the request and the rendering of a response. All responses are in JSON format except for the response for the home page which is HTML.

The Django application uses a MySQL database to store user session data as well as the references and details of uploaded images. To perform a matching process, the Django application uses the Python-MATLAB bridge described in Section 5.3 to start the job and receive the results for subsequent rendering.

[10]Django is an open source web framework written in Python. See `https://www.djangoproject.com`.

5.3 Python-MATLAB Bridge

The object recognition engine runs in MATLAB, however the web application runs in Python. Therefore a bridge between the two environments is needed for interaction. In essence, Python code needs to be able to call a MATLAB function and interpret the result.

Some solutions already exist for this purpose[11] however these all involve starting an instance of MATLAB for each function call. As the system should be fast and scalable, this is not a desirable solution, as there will be a 5-10 second overhead for each MATLAB function call while MATLAB starts up.

Instead, a Python-MATLAB bridge was achieved using MATLAB's Java runtime environment. Using some code written by D.Kroon (University of Twente), Java's TCP and socket libraries are used to create a TCP server to handle requests. This is run on a separate TCP port to the web application to avoid cross communication. This means that MATLAB needs to startup only once, after which function calls are issued with local HTTP requests. For example, to call the function `test_connect()` in the file `test_connect.m` with the MATLAB server running on port `4000`, simply issue an HTTP request to `http://localhost:4000/test_connect.m`. Arguments to the function can be added as query parameters to the HTTP request and decoded by the MATLAB function.

A MATLAB function `web_feval` with a supporting function `run_dot_m` simplifies this process so any MATLAB function can be called (whether it be on MATLAB's path or not) with any number of arguments, with the results returned as JSON structures. A Python class `Matlab` abstracts this on the Python side - any locally stored MATLAB function can be run, with arguments, and the results are returned as a Python dictionary. Listing 5.5 shows how a MATLAB function is called with the created class. The `run()` function takes two arguments: the location of the function and a dictionary of arguments to be passed in to the function.

Listing 5.5: Python code to run the object matching MATLAB function.

```
## Python
# First initialise the Matlab object. This is done once when the web server starts
matlab = Matlab('http://localhost:4000')
# Now run the desired function
while matlab.running:
       # Matlab is already processing something, wait
       time.sleep(2)
```

[11] See `http://claymore.engineer.gvsu.edu/~steriana/Python/pymat.html` for an example.

```
resp = matlab.run('~/Sites/4YP/visualindex/wikilist_dataset/demo_wiki_get_objects.m',
    {'image_path': query_image_path, 'display': 1, 'log_file': '%slogs/%s-log.txt' %
    (settings.MEDIA_ROOT, request.POST.get('key'))}, maxtime=999999)
# Get the results
result = resp['result']
```

With this bridge, running any MATLAB functions becomes trivial and so the web application can run the MATLAB matching process.

Chapter 6

Geometric and Descriptor Improvements

A number of modifications to the baseline system were explored to increase the matching performance of the object recognition system. These improvements are using the NOSAC algorithm in place of RANSAC (Section 6.1), using a different feature detector and transformation estimation algorithm based on affine invariant feature regions (Section 6.2), and modifying the descriptors to improve conditioning (Section 6.3).

Recalled from Chapter 4, the baseline system uses RANSAC to estimate an affine transformation between the query image and model images for spatial verification. If the number of corresponding features that conform to this transformation exceeds a threshold, the inlier threshold, the match is said to be spatially verified. Table 6.1 shows the impact on yield (the percent of objects in validation images successfully recognised) by implementing these improvements and varying the inlier threshold.

	Inlier Threshold			
	4	5	7	9
RANSAC	12.0%	14.1%	18.1%	14.3%
NOSAC	20.7%	20.8%	17.7%	16.1%
Affine	22.5%	22.3%	17.4%	13.8%
Affine RootSIFT	24.5%	22.6%	18.0%	15.5%

Table 6.1: The yield acheived for each improvement method with different thresholds for the number of RANSAC inliers required for a match.

6.1 NOSAC

To estimate an affine transformation between two images, six variables must be estimated (a, b, c, d, t_x, and t_y in Equation 4.4). Each feature point that is computed for an image contains four data points: the x coordinate, y coordinate, scale s, and rotation θ of the feature. The baseline RANSAC method randomly selects three correspondences and uses the x and y coordinates to yield two equations for each correspondence. For three correspondences, that means there are six equations, so the transformation can be solved. The best transformation (the one with the most inliers) over all the iterations of sampling is then used as a basis for the overall affine transformation.

This method works well, however it is based on random sampling so either accuracy or speed is sacrificed. Instead, a new estimation method is used called NOSAC.

Rather than selecting three correspondences, only one correspondence is needed to estimate a transformation. Initially, it is assumed the only components of the transformation are scaling and translation, as shown in Equation 6.1. There are therefore only three unknown variables - a, t_x, and t_y. The scaling factor a is estimated from the relative scales of the feature points, and the translation from the coordinates. An estimation is done for every correspondence. The inliers of the best estimation of the transformation are then used to compute the full six degree of freedom affine transformation[1].

$$\mathbf{x}' = \begin{bmatrix} a & 0 & t_x \\ 0 & a & t_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{x} \tag{6.1}$$

Listing 6.1: Estimating the pure scale affine transformation for each corresponding feature.

```
%% MATLAB
% for each correspondence
for i=1:n_correspondences
    % compute transformation (a 0 tx; 0 a ty; 0 0 1) where a = s2/s1
    a = f2(3,i)/f1(3,i);
    T = f2(1:2,i) - s*f1(1:2,i);
    H{i} = [s 0 T(1); 0 s T(2); 0 0 1];
    % score all the other points from transformation
    X2_ = H{i} * X1;
    delta = X2_ - X2;
    ok_rough{i} = sum(delta.*delta,1) < (1*thresh)^2;
```

[1] This is done using `vgg_Haffine_from_x_MLE` by Andrew Zisserman found on `http://www.robots.ox.ac.uk/~vgg`

```
    score_rough(i) = sum(ok_rough{i});
end
```

As an estimation is performed on every correspondence, there is no randomness to the process. Also, as the number of iterations is the number of correspondences, the NOSAC method is considerably faster than RANSAC. Matching performance is also increased by 15%.

6.2 Affine Invariant Detector

An alternative feature detector is used as an improvement. The Hessian affine region detector detects regions similar to the SIFT detector of the baseline system. However, the features detected are affine invariant - the detector will detect the same features and describe them in the same way even if the viewpoint has changed. This has the advantage of allowing matching of features even when the view of the object is markedly different.

The feature points are now six-dimensional rather than four-dimensional, with each feature region being described by its x and y position, orientation θ[2] and the ellipse representing the region, given by three parameters a, b and c such that the ellipse is described by Equation 6.2.

$$ax^2 + 2bxy + cy^2 = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{x^T S x} = 0 \tag{6.2}$$

There are two advantages of using the affine invariant detector. The first is that there are more corresponding features between images of the same object as the a feature from a different viewpoint is always detected, and described in the same way.

The second is that better estimations of the affine transformation can be done during the NOSAC process. However, rather than assuming purely scaling and translation as in Section 6.1, a full affine transformation can be computed for each correspondence. This is because each feature is described by an ellipse. For a correspondence between two points with ellipses described by

$$\mathbf{x_1^T S_1 x_1} = 0 \tag{6.3}$$

[2]In practice, no useful interpretation of the orientation has been found.

and

$$\mathbf{x_2^T S_2 x_2} = 0 \tag{6.4}$$

related by

$$\mathbf{x_2} = \mathbf{H x_1} \tag{6.5}$$

the unknown affine transformation $\mathbf{H}$ can be found by solving

$$\mathbf{S_1} = \mathbf{H^T S_2 H} \tag{6.6}$$

As with NOSAC, this is done for every correspondence and the inliers from the best estimation used to compute a final estimate for the transformation. The process is faster than RANSAC and results in a 24% increase in yield.

[INCLUDE REMOVAL OF REPEATING STRUCTURES SOMEWHERE???]

6.3 RootSIFT

To measure the similarity between two features, the Euclidean distance between the SIFT descriptors is used in the baseline system. However, a recent publication [2] notes that using a different distance measure can increase performance.

The Hellinger kernel is used in place of the standard Euclidean kernel for computing distance. This change can be performed by simply mapping the descriptors from SIFT space to RootSIFT space - the RootSIFT descriptor is an element wise square root of the L1 normalised SIFT vector. Computing the Euclidean distance between two RootSIFT vectors is equivalent to using the Hellinger kernel for comparing the original SIFT descriptors. Listing 6.2 shows the conversion of the SIFT descriptor `sift` to RootSIFT space.

Listing 6.2: Mapping of SIFT descriptors to RootSIFT space.

```
%% MATLAB
root_sift = sqrt(sift/sum(sift));
```

The effect of using RootSIFT is to reduce the larger SIFT dimensions in relation to the smaller ones. This prevents the Euclidean distance being dominated by these large values, which could be noisy. By using

RootSIFT space with the affine invariant detector, matching performance is increased 35% compared to the baseline system

Chapter 7

Augmentation Improvements

This chapter describes the improvements made using a technique to inflate the visual word content of the model images in the database. This novel technique, defined as *turbo-boosting*, increases the performance by xyz as shown in Table HAHAHAH.

7.1 Motivation

The performance of the baseline system with the improvements described in the previous chapter is good. However, it can be argued that the yield the object recognition engine can achieve is limited by the amount of accurate image data.

A data source such as Wikipedia is very good at providing labelled images - at least one of the images on an article page will depict the object of the article. However, the other images may be related to esoteric aspects of the article and may not actually depict the object in question. This is advantageous in one aspect as it allows esoteric matching of the object. Unfortunately, this also results in the amount of specific image data for an object being very low. In many cases there are only one or two images of the actual object in question. For example, for the object `10_Downing_Street`, there are 16 images in total, though only two show the front facade of the building. All the validation images are of the front facade of 10 Downing Street, as that is what one would associate most with that object.

WIthout more images of the same object, viewpoints, environmental conditions, objects of occlusion, and noise content of individual images may vary to such a large extent that an obvious match cannot be

Figure 7.1: A selection of images from Wikipedia for the object `10_Downing_Street`.

made.

7.2 Database Augmentation

Existing work addresses this problem by including a process known as database augmentation. This database-side feature augmentation works by including the visual words present in other images of the same object for an image of an object. For each model image, a query using the baseline system is issued and all the spatially verified matches are collected. In the original database augmentation technique described in [1], the visual words of the model image used as the query are augmented with all the visual words of the spatially verified database matches. Further work such as [2] shows that instead of incorporating the entire vector of visual words from database matches, only the visual words contained within the regions in the database matches that correspond spatially are augmented.

[Figure of database-side augmentation?]

The process of querying the baseline system with each model image creates what is known as an image graph - the interconnection of matches between images. Each edge of the image graph infers that the two images connected share (at least part of) the same object. An image graph is not needed in this project, as

the absolute classes of each image are already known, and it is assumed that overlap between classes does not exist[1]. In dataset of model images used in this project, the lack of multiple images of the same object means that the amount of augmentation that occurs is negligible, so the gains of database-side spatial feature augmentation is non-existant.

7.3 Turbo-boosting

While database-side feature augmentation is not suited to this application, the overall idea of feature augmentation can still be used. Rather than database-side feature expansion, an independent source of images is used for augmentation. The turbo-boosting can be thought of as expanding the information content of your ground-truth image database using an external dataset.

The external source of images used for turbo-boosting in this project is Microsoft's Bing image search, though any labelled image dataset can be used. The rationale for using a search engine is that labelled images for each class (a class being an object that could be contained with in an image) can be generated by issuing a search query. Therefore, for images of a class, the candidate images to use for augmentation are the results of a search query of the class name. The result is that turbo-boosting keeps the accuracy of the ground truth of your dataset but expands the information content of the dataset images.

Listing 7.1 shows the high level MATLAB code that outlines the implementation of turbo-boosting. A class centric approach is taken, so the images from each class are augmented in the same loop. For each class, the candidate images for augmentation are downloaded, called *turbo images*. Section 3.2 describes the data gathering process in detail. Then, for each model image belonging to the class in question, each downloaded turbo image is tested to see if it can be spatially verified against the model image. If a turbo image is spatially verified, a rectangle that is the bounding box of the spatially verified visual words is computed (with a margin of tolerance). All the visual words contained within the rectangle on the turbo

[1] This is a contentious point. Due to the nature of images from Wikipedia, there is a fair amount of overlap between images between classes. For example, a picture of Big Ben may well be included in the corpus of images for Palace of Westminster. This is an issue that produces false positives. Methods to remedy this issue can be employed, such as querying the baseline system with each model image, and if a false positive match is made, remove the image. However, this is beyond the scope of this report.

image are added to the visual words in the model image at the position mapped by the affine transformation relating the two images.

Listing 7.1: High level MATLAB code to outline the augmentation stage of the turbo-boosting process.

```
%% MATLAB
% Get a list of classes
classes = get_class_names(db);
for i=1:length(classes)
    class = classes(i);
    % Download images from an external source for that class
    turbo_images = download_images(class);
    % Get the model images for that class
    model_images = get_images_by_class(class, db);
    for j=1:length(model_images)
        model_image = model_images(j);
        model_im_words = get_words(model_image, vocab);
        for k=1:length(turbo_images)
            turbo_image = turbo_images(k);
            turbo_im_words = get_words(turbo_image, vocab);
            % Get corresponding visual words
            correspondences = get_corresponding_words(model_im_words, turbo_im_words);
            % Spatially verify correspondences
            verified, verified_model_words, verified_turbo_words, affine_transformation =
                spatially_verify(correspondences);
            if verified
                % Get the region within the turbo_image that contains
                % words
                bounding_rectangle = get_bounding_box(verified_turbo_words);
                verified_turbo_words = words_within_rectangle(bounding_rectange,
                    verified_turbo_words);
                % Map the verified turbo words within the bounding
                % rectangle onto the model image and augment these words
                model_im_words = augment_words(model_im_words, verified_turbo_words,
                    affine_transformation);
            end
        end
        % Save the fully augmented word vector for the model image
        save_augmented_words(model_im_words);
    end
end
```

The process inflates the the visual word vector used for spatial verification on the fly - as the for-loop continues with each subsequent turbo image to boost a model image, the model image's visual word vector gets larger and so there is more chance for further spatial verification and therefore augmentation.

Once the turbo-boosting for every model image in every class has completed, the weighted histograms and new turbo-boosted index is created, just as for the baseline system (see Section 4.3). Matching is then performed just as with the baseline system, except that the turbo-boosted index and visual word vectors are used.

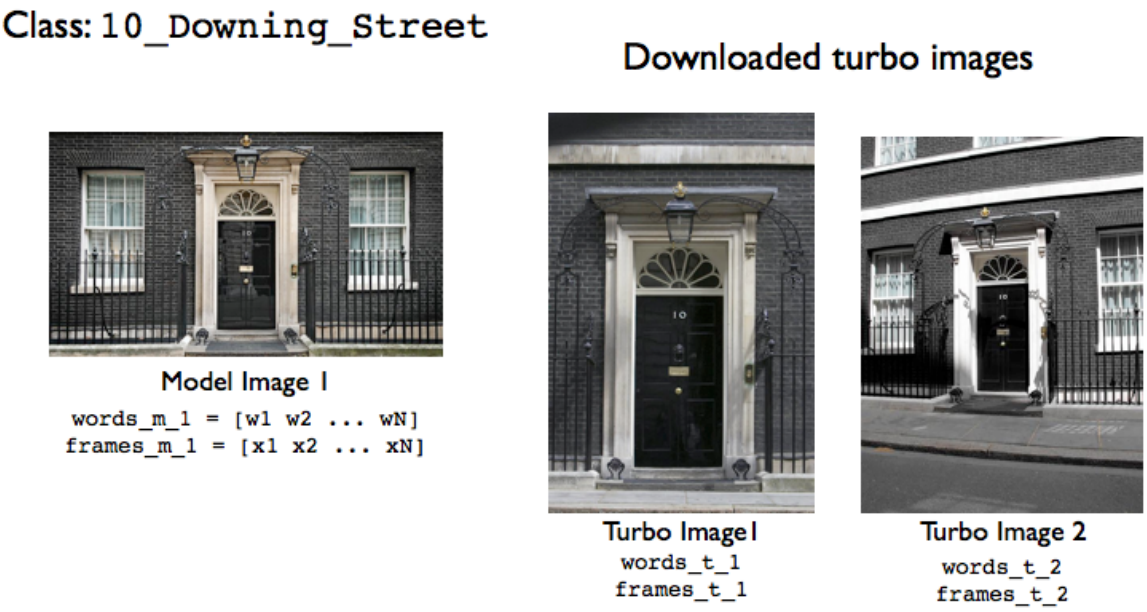
The results of turbo-boosting with different sized candidate pools of turbo images is shown in Table

woop woop. Matching performance is increased x% over the equivalent system not employing turbo-boosting.

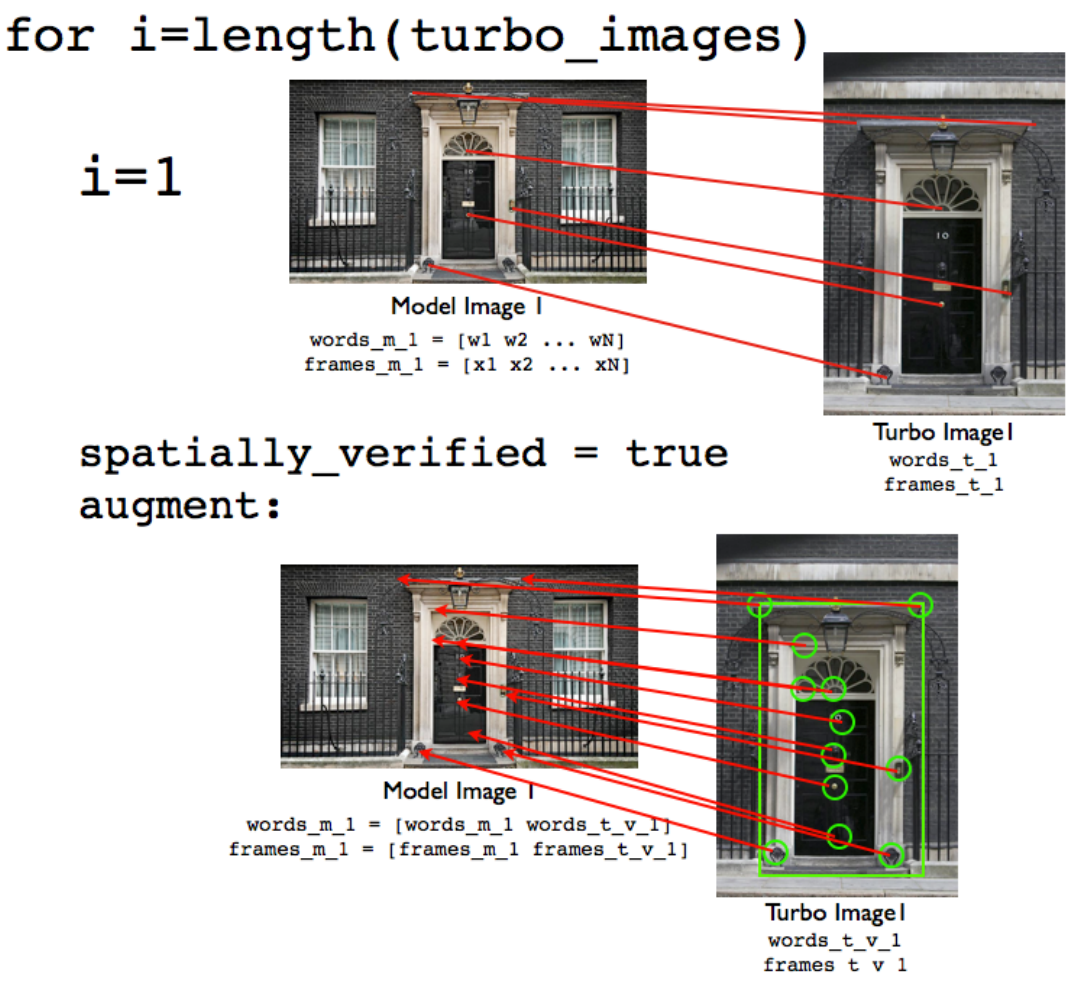
Turbo-boosting works by increasing the information content of the model images. When a turbo image is spatially verified with a model image, it may be verified on a minimum seven visual words. However, many hundreds of words can be added to the model image's visual word vector, as all the words contained within the bounding box of the few spatially verified visual words are added. These added words help matching in three ways:

- There may be features that were detected in the turbo image that were not detected in the model image. When these visual words are added to the model image from turbo-boosting, this increases the detailed description of the object in the model image.
- Some features common to both images may be described by different visual words due to noise. After turbo-boosting, both visual words will therefore be in the same position in the model image describing the same feature. This increases the robustness to noise.
- Some augmented features will be duplicates - a duplicate feature being one that is common to both the turbo image and the model image and is described by the same visual word in both cases. This means that after turbo-boosting, the model image will contain multiple identical visual words for the same feature. This effectively gives the feature more weight for matching.

Turbo-boosting is a completely general strategy for increasing the yield/recall - any labelled dataset can use this technique. The only draw backs are the increase in pre-computation time due to the turbo-boosting routine, which is $O(N_c\overline{N}_m N_t^2)$ where N_c is the number of classes, $\overline{N}_m$ is the average number of model images of each class, and N_t is the number of turbo images that are downloaded. Although, the process is easily parallelised as the inner loop on each class is completely independent. Another disadvantage is the slight increase in storage needed. The downloaded turbo images can be deleted as they are only needed temporarily, however due to the increase of visual words in each model's vector and the fact that the histograms will be generally less sparse after augmentation, the byte size of these structures increases.



(a)



(b)

Figure 7.2: The turboboosting process [I NEED TO MAKE THIS MUCH BETTER].

Chapter 8

Summary

[Removal of duplicates + ambiguous class objects]

Bibliography

[1] P. Turcot and D. G. Lowe. Better matching with fewer features: The selection of useful features in large database recognition problems In *WS-LAVD, ICCV*, 2009.

[2] R. Arandjelović and A. Zisserman. Three things everyone should know to improve object retrieval In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012