The Intelligent Image

Max Jaderberg

Keble College, University of Oxford

Trinity Term 2012

Abstract

In this report, a large-scale object retrieval system is presented. It allows the user to supply a query image via a web interface, and the web site returns the *intelligent image* – the objects in the image are automatically labelled and can be clicked for more information.

The system builds upon the state-of-the art in visual object retrieval from large databases – that is allowing objects to be recognised by searching against millions of images. This notion is explored by using the images contained on the web pages of an online encyclopaedia as the object database. The resulting system can automatically annotate multiple objects within query images, such as the landmarks in a holiday photo. We demonstrated that the system can be implemented with a front end web user interface for interaction. A number of improvements to the base line system are investigated, including a novel method for increasing the matching performance by external database augmentation.

This work is done with the view to scaling up to the database to a corpus of millions of images and objects.

Contents

1 Introduction **3**

1.1 Motivation 3

1.2 The Intelligent Image 4

1.3 Road Map 5

2 Background **6**

3 Data **9**

3.1 Model Images 9

3.1.1 Data Acquisition 10

3.2 Augmentation Images 12

3.2.1 Data Acquisition 12

3.3 Test Images 14

3.3.1 Data Acquisition 15

3.3.2 Performance Measure 16

4 Baseline system **18**

4.1 Feature Detection and Description 20

4.2 Visual Words 20

4.3 Histograms and Index 21

4.4 Spatial Verification 22

4.5 Multiple Object Matching 24

4.6 Performance 24

5 Geometric and Descriptor Improvements **25**

5.1 NOSAC 25

5.2 Affine Invariant Detector 27

5.3 RootSIFT 28

5.4 Results 30

6 Augmentation Improvements **31**

6.1 Motivation 31

6.2 Database Augmentation 32

6.3 Turbo-boosting 32

6.4 Results and Analysis 35

7 Implementation **38**

7.1 Object Recognition Engine 38

7.1.1 Database 39

7.1.2 Image Processing 40

7.1.3 Distributed Pre-computation 40

7.2 Web Application 41

7.2.1 Front-end 42

7.2.2 Back-end 43

7.3 Python-MATLAB Bridge 44

7.4 Examples 45

8 Summary **46**

Chapter 1

Introduction

1.1 Motivation

The web contains billions of images, and they 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 their containing HTML[1] code. Very little information is available about the images, let alone the objects or scenes contained within them. The aim of this project is to create a system which automatically recognises the objects within images, thus releasing the information within them. This is a large scale object retrieval problem.

Recognising a wide range of objects in an image is not a trivial problem. Objects can be photographed from many different viewpoints, under different lighting conditions, with different cameras, and can be occluded in different ways. This presents problems of robustness to these types of variation and noise. As more objects are added to the database, scaling the solution up becomes a problem. Recognition accuracy drops as there are more objects that could be potential matches, and the matching process gets slower as a larger search is needed. These challenges were addressed throughout the project to ensure a fast, accurate recognition system that can be scaled up to millions of objects.

There are a large number of applications that 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

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

(a)

(b)

(c)

Figure 1.1: The standard image Figure 1.1a and the intelligent version of the image Figure 1.1b with it's hover over state Figure 1.1c.

was photographed. The plethora of useful applications provides a great deal of motivation for this project.

1.2 The Intelligent Image

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 that knows about the contained scene and can offer up information to the consumer about it's contents.

The tagging process works by matching a section of the query image with an image of an object from a database. The query image is then said to contain this object within the region that matches. To be able to recognise a vast corpus of objects, a large collection of reference images is needed to be available to match against. The database of images of objects is created from Wikipedia [19] – a crowd-sourced online encyclopaedia with many images contained within the articles. Each Wikipedia article defines an object, and the images that appear in the article are downloaded to the database. If a query image matches a

Wikipedia image in the database, the query image is tagged as the object defined by the article from where the Wikipedia image was downloaded.

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 of objects chosen are the pages that appear on the Wikipedia page "List of Structures in London" [20].

1.3 Road Map

The report starts by introducing the background information are previous work published in Chapter 2. This gives a holistic view of the recent innovations in large scale object recognition and retrieval and provides the basis of knowledge used in the project.

Chapter 3 gives an in-depth account of the data sources used in the project. This includes the acquisition of the database of Wikipedia images as well as various other datasets used for testing and improving the system.

Chapter 4 explains the workings of the baseline system. In this chapter, the steps required for the basic tagging process to work are described. These include the representation of images as collections of *visual words*, the matching of query images to database images by their visual word similarity, and the spatial verification that occurs to confirm a match.

Chapters 5 and 6 explain the research and development of improvements to the baseline system to increase object recognition performance. Improvements to the spatial verification stage are made and new feature detectors are implemented. These increase the speed and accuracy of matching. Another novel method was developed to further improve matching performance by augmenting the images in the database with those of an external source.

Finally, Chapter 7 describes the practical implementation of the object recognition system, as well as the development of a web site to provide a front end user interface to create an intelligent image.

Chapter 2

Background

This project builds upon the vast amount of previous work done on large scale image retrieval and automatic image annotation.

There is no general algorithm to determine whether a query image[1] contains the same object as an image in a dataset – the variation in viewpoint, lighting, environment and noise is so large that this task can even be a struggle for humans. However, despite this absence there has been a large amount of work devoted to creating applicable solutions to this problem [1, 4, 7, 9, 11]. Types of recognition range from scene classification [15] to geo-location recognition [16], however it is specific object annotation that this project aims to achieve.

To be able to query images, a representation of images needs to be computed such that some evaluation of similarity can be applied. All current methods use the notion of detecting salient regions of an image, *features*, and describing them by some vector, a *descriptor*, then using the distance between feature descriptors as a measure of similarity. Basic feature detectors could be corner detectors (such as the Harris detector) and a descriptor could simply be a vector of the local grayscale values, however a method like this is prone to variance due to changes in viewpoints and lighting. It is the work on SIFT descriptors [10] that is used in this project, as this detector and descriptor is invariant to many of the distortions that occur in images of the same object.

The standard method for image retrieval that is used is the bag-of-words method described in [7, 9]. The approach by Zisserman *et al.* in [9] is closely followed and described fully in Chapter 4. This method involves quantisation of descriptors into "visual words"[2] that are used for matching. The paper also

[1] A query image is one that contains a depiction of an object that is to be identified.

[2] The term "words" in the context of image retrieval will refer to this notion of "visual words".

describes the influence of work done on web search engines in retrieving useful web pages – the established methods for document retrieval from text queries described in [5, 6] are adapted for use with visual words. A weighting is applied to each visual words to signify their importance in retrieval (the tf-idf weighting), and inverted file structures are used to provide fast lookup and filtering of the images in the corpus.

Systems such as [11] have improved recognition performance by making use of the spatial information associated with visual words. Philbin's system performs a re-ranking of the matches based on visual word similarity by estimating an affine transformation between the features in the images of matches and the query image. This ensures that that the objects in a proposed match are structurally similar – an object in an image is in essence defined not only by the visual words it exhibits, but the positions and scale of those visual words within an object's image. Making use of spatial information was shown to consistently improve performance by 5%.

Other improvements in image retrieval have come from advancements in clustering methods. Early systems [9] used a flat k-means clustering on feature descriptors to create the visual words, but this is difficult to scale. Cluster hierarchies [12] allow the size of vocabularies to be greatly increased, and [11] demonstrates that an approximate nearest neighbours method works best. Traditional k-means uses a vast amount of computational power on calculating the nearest neighbours between points and cluster centres, however an approximate method can be used instead by employing a k-d tree and a random forest algorithm [13, 8, 3]. The result is that far superior performance can be achieved using very large vocabularies (100K words and greater).

The paper by Quack *et al.* [1] describes the application of a state-of-the art system to provide automatic annotation of landmarks in images. A database of images with associated metadata is built by crawling Wikipedia and other internet sources. This is built from the content of a previous paper [17], where photos are mined by geographic proximity from a photo sharing website. The resulting database is clustered into objects, and the images from Wikipedia articles are used for verifying the cluster assignments and provide the labelling. Retrieval is then performed using the bag-of-words approach with spatial verification. Quack *et al.* also use a method of identifying the useful visual words for a cluster of images of objects, allowing the bounding box of the object to be determined as well as facilitating the reduction in the size

of the database. However, this system uses the geographical location data associated with photos to allow intelligent clustering and filtering to be done to ensure accurate object representations. This reduces the complexity of the problem compared to the system in this paper where no geographical data is available, but similar performance needs to be achieved.

There are other improvements that have been developed, such as database-side feature augmentation. This method described in [2] is a way of increasing recognition performance by disseminating the visual word information across the database images. For each image in the database, a query is issued and the spatially verified matches are collected. The visual words of the matches are then augmented with the visual words of the database query image. Further work such as [4] 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.

The state of the art as described in [11] is used as the basis for this project, with the other research mentioned used for inspiration to enhance the performance.

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). These are referred to as *model images* and are used in Chapter 4 when creating the baseline system.
- The images from Microsoft Bing that are used for the augmentation improvements. A description of this dataset is found in Section 3.2, and their usage is in Chapter 6.
- Test images gathered from Google Images, described in Section 3.3 and used in Section 3.3.2.

Table 3.1 gives an overview of the data used. 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.

	Source	# Images	# Classes
Model images	Wikipedia	3963	732
Augmentation images	Microsoft Bing	18273	732
Test 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. As explained fully in Chapter 4, an object is recognised by matching a sub-part of the query image containing an object to an image of the object stored in a database. This database of images of objects is created

from the images from Wikipedia, and the images downloaded from Wikipedia for this database are known as *model images*.

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.

The subset of Wikipedia articles used in this project are the articles containing images linked to by the "List of Structures in London" page [20]. This results in 732 unique objects for the database comprising of 3923 images (on average 6 images per class). The images are pulled directly from the articles for the objects. [22] is a list of all the classes used in the application.

Figure 3.1 illustrates some of the object classes and the model images downloaded for them.

Figure 3.1: A selection of some object classes with the model images associated with them.

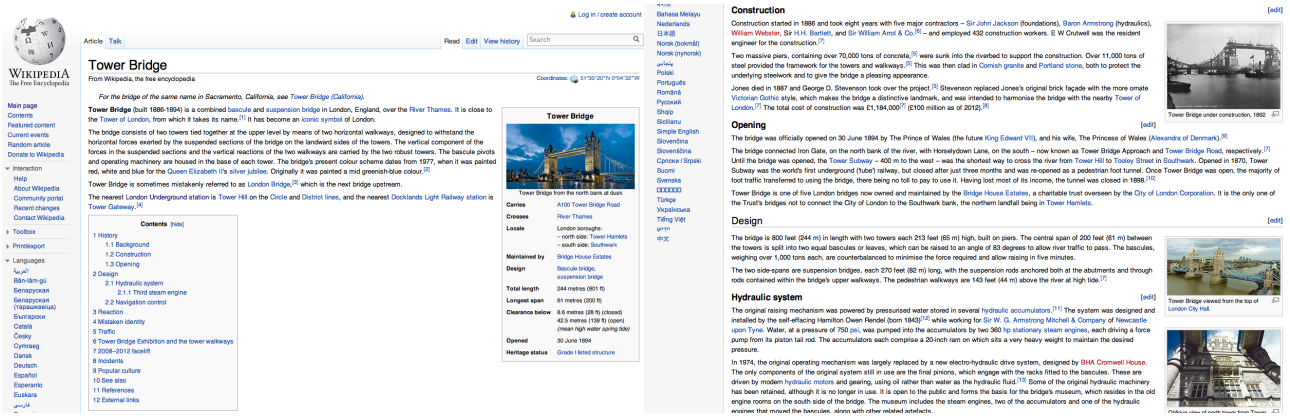
3.1.1 Data Acquisition

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.

[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.



(a) (b)

Figure 3.2: The Wikipedia page for "Tower Bridge". Note the images contained are those used in the model to represent this object.

Firstly, the HTML of the Wikipedia page must be downloaded, as it appears to a web browser. 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 HTML document for each Wikipedia page is parsed using the BeautifulSoup library[5]. All the anchor elements are found and stored for further 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.

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 image and later resize it. Instead, Wikipedia's inbuilt thumbnail engine is exploited, which resizes the image

[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/`

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

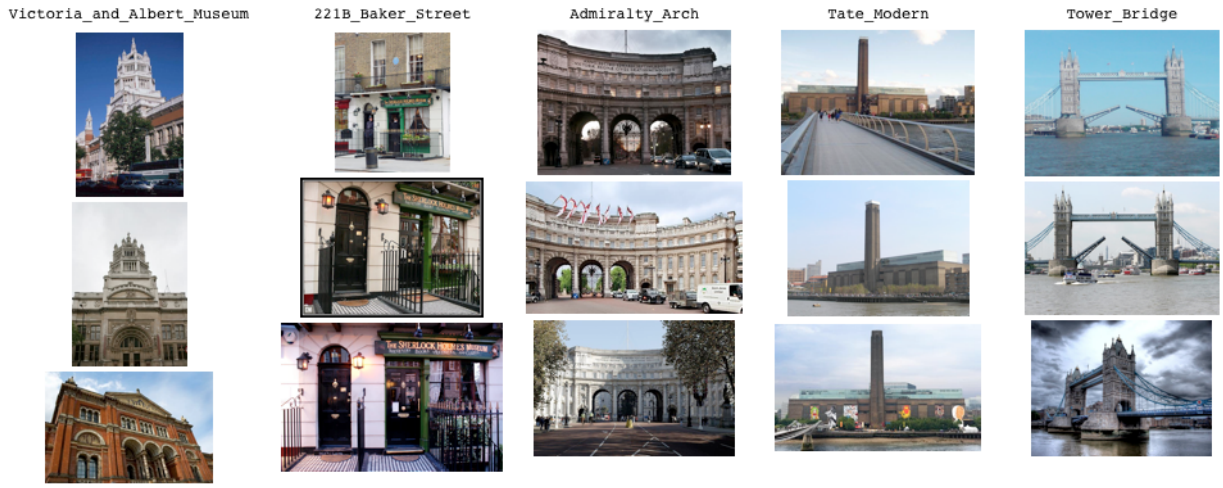


Figure 3.3: A selection of the augmentation images for some object classes.

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.

3.2 Augmentation Images

One of the methods of improving the performance of the baseline system is by augmenting the model images with the data from an external source of images. This process, described in Chapter 6, requires a dataset of images to use for augmentation. This is the focus of this section.

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

Overall, the dataset contains 18273 images distributed over the 732 object classes. A selection of the augmentation images dataset is shown in Figure 3.3.

3.2.1 Data Acquisition

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].

[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.

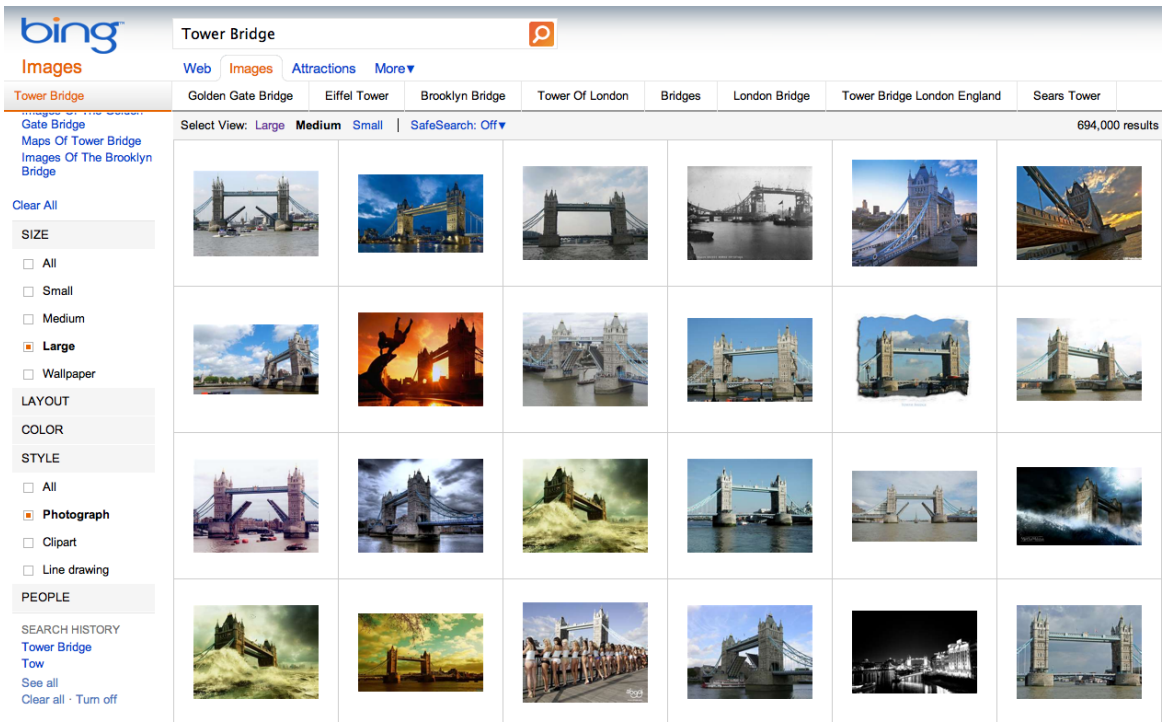


Figure 3.4: The browser interface of Bing image search which is replicated in the API query URL. Note the filters in the left column.

Bing image search takes a number of keywords – the query – and returns a list of images from web pages related to the query. Filters can be applied to further narrow down the search to the most relevant images. This is shown in Figure 3.4 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 augmentation.

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 3.1 shows the URL used for the API call to get the search results for the images for a particular 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 3.1 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 3.1: A Bing image search API request.

```
%% MATLAB
```

[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.

```
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);
```

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. 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.

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

3.3 Test Images

Images are needed to test and quantify the yield performance of the object recognition system (see Subsection 3.3.2). 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 is used first to automatically download 8 images for each class. However, many of the automatically downloaded test images are inaccurate representations of the object in question, or are completely unrelated due to a generic search term giving results that do not even show the object. For example, a search for the object "Foyles" (a book shop in London), gives some representative images of the shopfront, but there are also many images of logos, books, people and the inside of the store. As what is being tested is whether the *structure* of Foyles book shop can be recognised, these other images

Figure 3.5: A selection of some of the test images for objects.

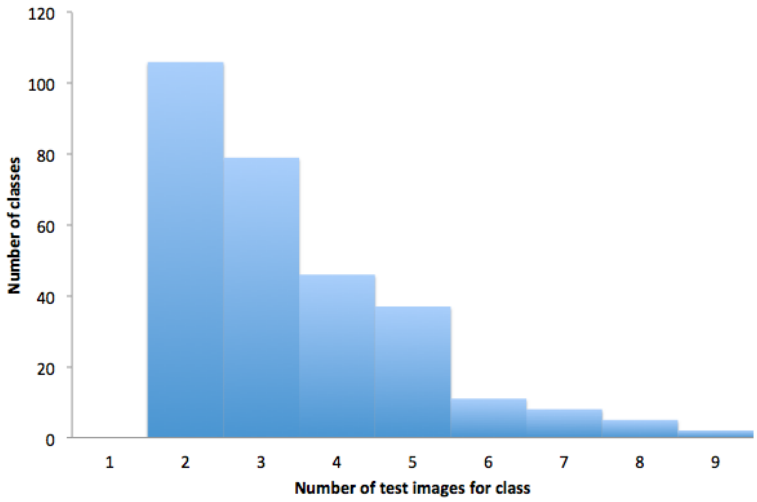


Figure 3.6: A histogram of the number of test images per class.

should be discarded. Another example is the search for "55 Broadway" gives some images of the building 55 Broadway in New York City, rather than London. Therefore, after automatic download, the dataset is manually refined to remove unrepresentative images.

The resulting test images dataset totals 701 images, spread out over 294 object classes. The distribution of test images is shown in Figure 3.6 which shows the histogram of how many classes contain a number of test images. Some examples of test images are shown in Figure 3.5.

3.3.1 Data Acquisition

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 that 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 3.2: 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 3.2 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 is parsed into a MATLAB object and the `url` field is used to download the image.

As with the augmentation images, the downloaded Google images are resized if larger than 1000 pixels and stored in a folder named after its class. Again, the result is a directory containing a folder for each class, within which are the test 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.

3.3.2 Performance Measure

The test images are used to measure the performance of the system. A query is issued for every test image and the label assigned by the system is compared to the ground-truth object label of the test image. The performance is measured simply as the ratio of test images that are correctly labelled by the system, to the total number of test images, expressed as a percentage. This is defined as the *yield*, and is described by

$$yield = \frac{1}{N}\sum_{i=1}^{N}[y_i = f(x_i)] \tag{3.1}$$

where y_i is the ground-truth class of image x_i, $f(x_i)$ is the class given by the system, and N is the total number of test images (701 in this case). For example, a yield of 30% means that 210 of the 701 test

images were annotated correctly. It should be noted that each test image is only associated with one ground-truth class[10].

With this measure of performance, the performance of the baseline system (Chapter 4) can be quantified, and the effects of the improvements described in Chapter 5 and Chapter 6 can inpterpreted.

[10]This means to say that a test image cannot contain multiple objects. There is however a problem during testing of some test images being matched accurately with model images, but their classes being different due to the labelling process. For example, a test image of Big Ben may be labelled as Houses of Parliament – this would be recorded as a false positive label, even though this would be a perfectly acceptable label for the test image. The result is some false positive labelling which is not actually incorrect. Therefore the true yield performances are always slightly higher than given due to some incorrect false positive labelling during testing.

Chapter 4

Baseline system

The baseline system draws upon the research and literature mentioned in Chapter 2. Upon starting the project, an application was provided written in MATLAB and C. However it simply ranked the similarity of database images to a query image, and the database was one of structures in Oxford that has been well researched. There was therefore a very large amount of work to take the starter application forward to the full system proposed by this project.

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.

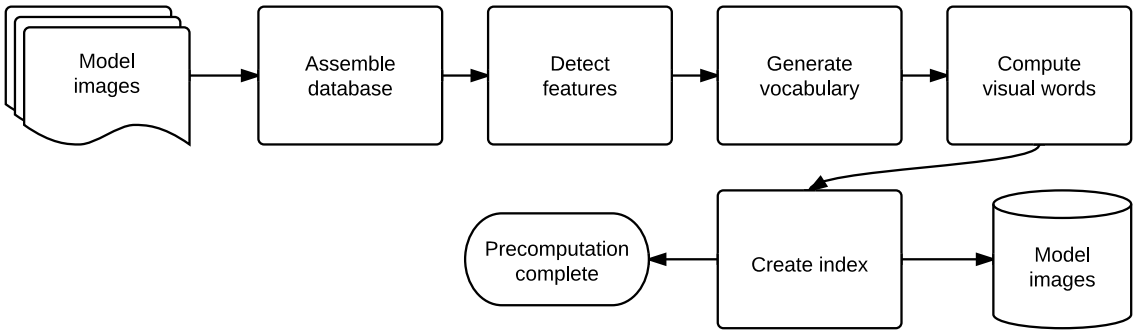


Figure 4.1: The flow diagram for the basic pre-computation process.

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

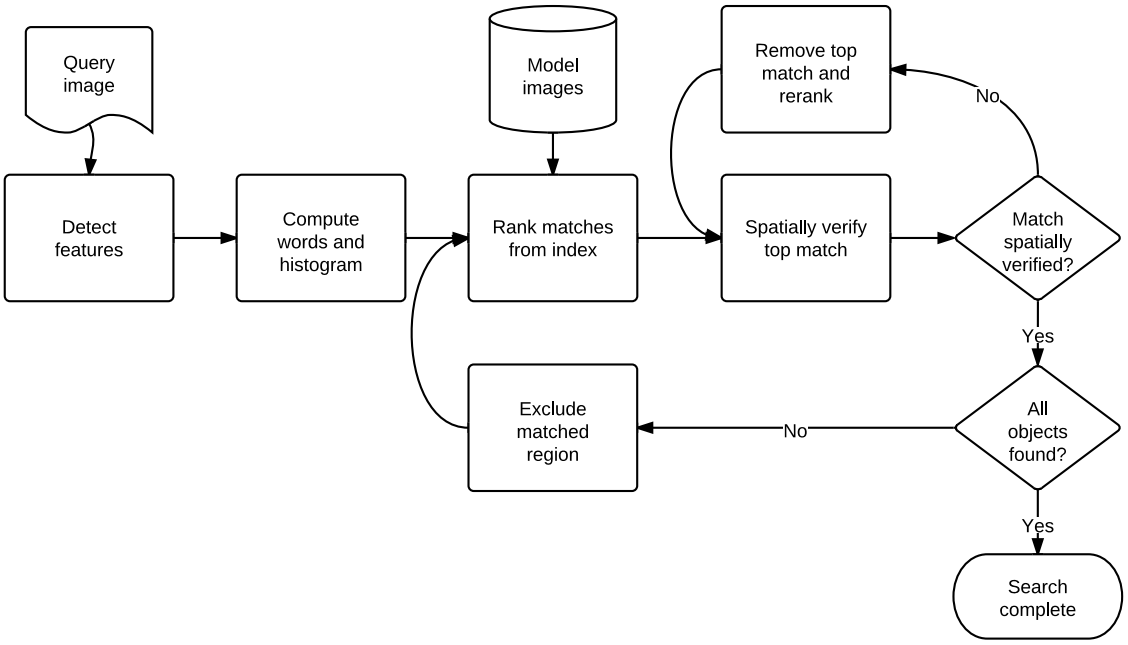


Figure 4.2: The flow diagram for the basic object recognition process.

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 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 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,

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

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, where each column is the 128-D vector $\mathbf{d_i}$ that describes feature i:

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

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,

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

100K 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 100K element array. Two examples of the raw histograms are shown in Figure 4.3b and Figure 4.3e.

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.

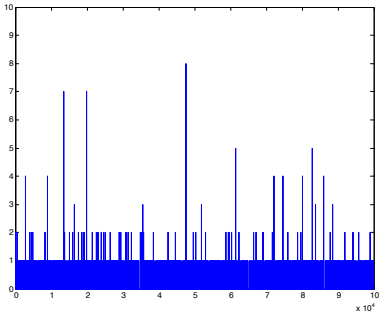
To find the most similar images to a query image, a dot product is performed between each of the model image histograms and the query image histogram,

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

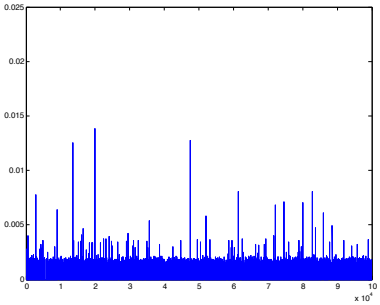
where $\mathbf{h_{query}}$ is the tf-idf weighted histogram of the query image and $\mathbf{h_i}$ is the tf-idf weighted histogram of model image i. $\mathbf{scores}$ is a $N \times 1$ array with the result of the dot product with each model image's histogram in each element. The matches are ranked in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

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

(a) Tower Bridge image 1

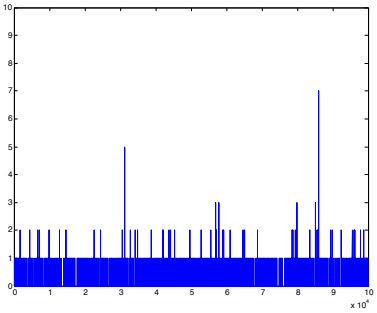


(b) Raw histogram for image 1

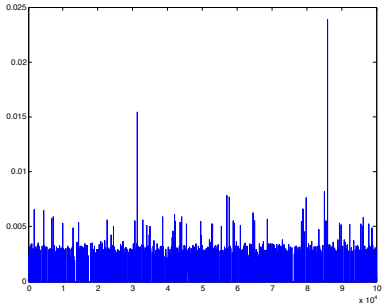


(c) Weighted histogram for image 1

(d) Tower Bridge image 2



(e) Raw histogram for image 2



(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

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.

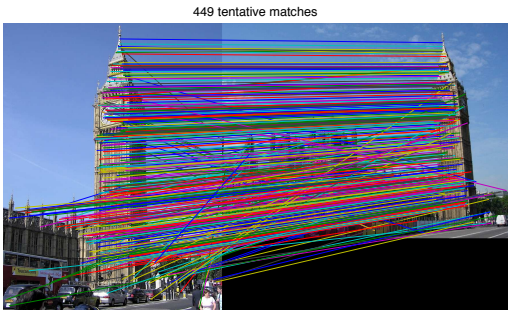
The spatial verification restraint in this application is that there should exist an affine transformation between the scene in the query image and the scene in the model image that is the potential match. Practically, this means that there should be an affine transformation that maps the visual words 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.

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

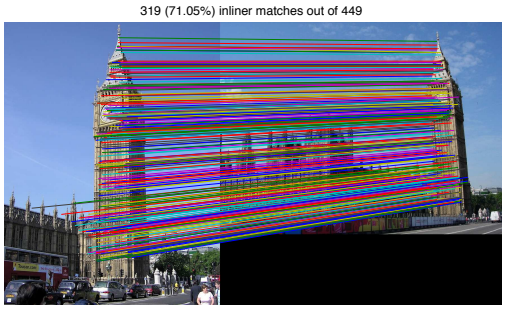
The random sample consensus algorithm (RANSAC) is used to estimate an affine transformation between the two images, using the corresponding visual words in each image. All duplicate visual words are removed from the spatial verification process, as these duplicates often cause inaccurate estimations. The result of the 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.

(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 4.4 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.

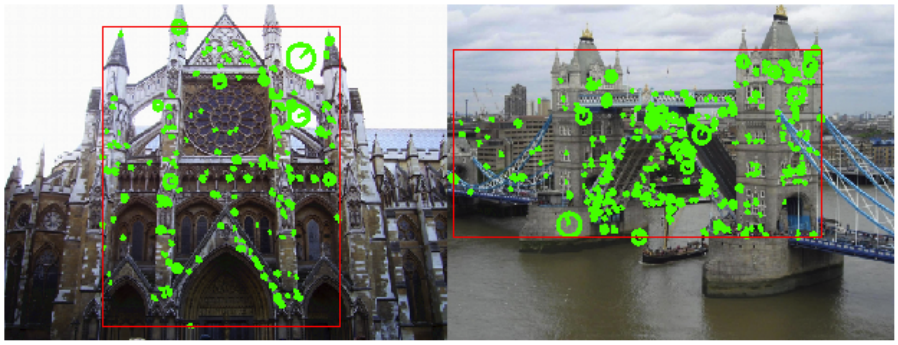


Figure 4.5: An artificially created image showcasing multiple object matching.

4.6 Performance

The baseline system was tested to obtain a measure of the object recognition performance. The best achievable yield (see Section 3.3.2) was 18.1%. This is fairly low for this sort of application. There are a number of limitations in the matching process, such as with the spatial verification, as well as the lack of representative images that are the root cause of this performance.

Chapter 5

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 5.1), using a different feature detector and transformation estimation algorithm based on affine invariant feature regions (Section 5.2), and modifying the descriptors to improve conditioning (Section 5.3).

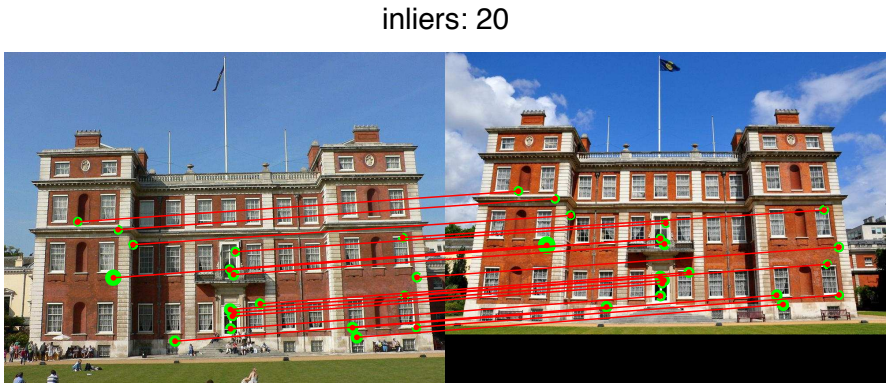
5.1 NOSAC

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 5.1. 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

(a) RANSAC



(b) NOSAC

Figure 5.1: The difference in matching by using NOSAC compared to RANSAC

then used to compute the full six degree of freedom affine transformation[1].

$$\mathbf{x}' = \begin{bmatrix} a & 0 & t_x \\ 0 & a & t_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{x} \tag{5.1}$$

Listing 5.1: Estimating the pure scale affine transformation for each corresponding feature.

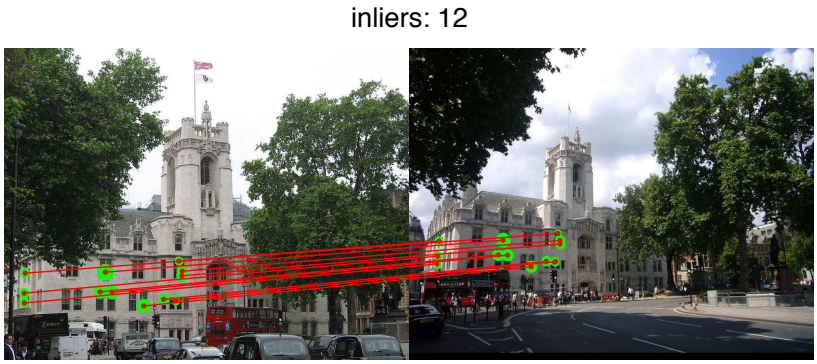
```
%% MATLAB
% for each correspondence
for i=1:n_correspondences
    % compute transformation (a 0 tx; 0 a ty; 0 0 1) where a = s2/s1
    a = f2(3,i)/f1(3,i);
    T = f2(1:2,i) - s*f1(1:2,i);
    H{i} = [s 0 T(1); 0 s T(2); 0 0 1];
    % score all the other points from transformation
    X2_ = H{i}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

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

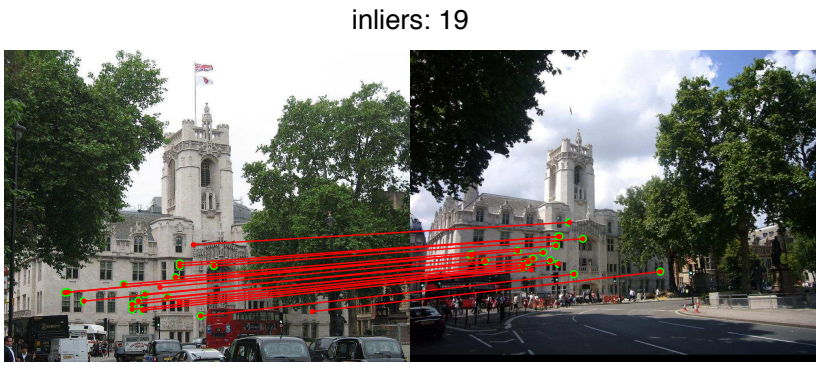
As an estimation is performed on every correspondence, there is no randomness to the process. Also, as the number of iterations is the number of correspondences, the NOSAC method is considerably faster than RANSAC. Matching performance is increased by 15% compared to that of the baseline system, to an absolute yield of 20.8%.

5.2 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 viewpoint of the object is markedly different.



(a) NOSAC



(b) Affine Invariant

Figure 5.2: The difference in matching by using the affine invariant feature detector compared to NOSAC.

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 and the ellipse representing the region, given by three parameters a, b and c such that the ellipse is described by

$$ax^2 + 2bxy + cy^2 = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{x^T S x} = 0. \tag{5.2}$$

There are two advantages of using the affine invariant detector. The first is that there are more corresponding visual words between images of the same object as the features from a different viewpoints are 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 5.1, 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

$$\mathbf{x_1^T S_1 x_1} = 0 \tag{5.3}$$

and

$$\mathbf{x_2^T S_2 x_2} = 0 \tag{5.4}$$

related by

$$\mathbf{x_2} = \mathbf{H x_1} \tag{5.5}$$

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

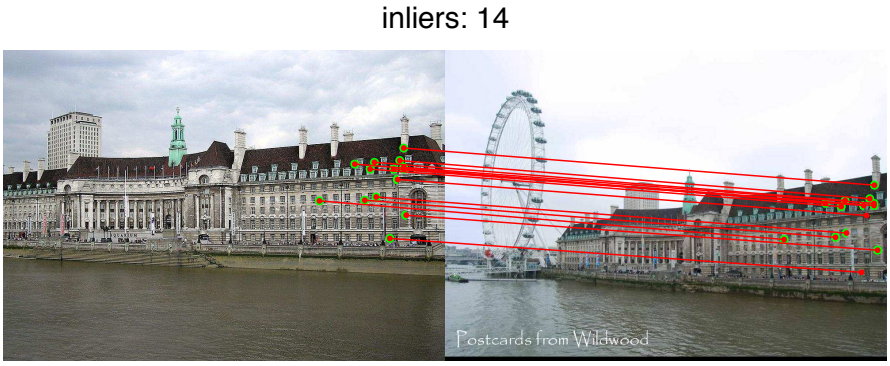
$$\mathbf{S_1} = \mathbf{H^T S_2 H} \tag{5.6}$$

given the constraint that vertical is mapped to vertical (see Section 4.1 in [11]).

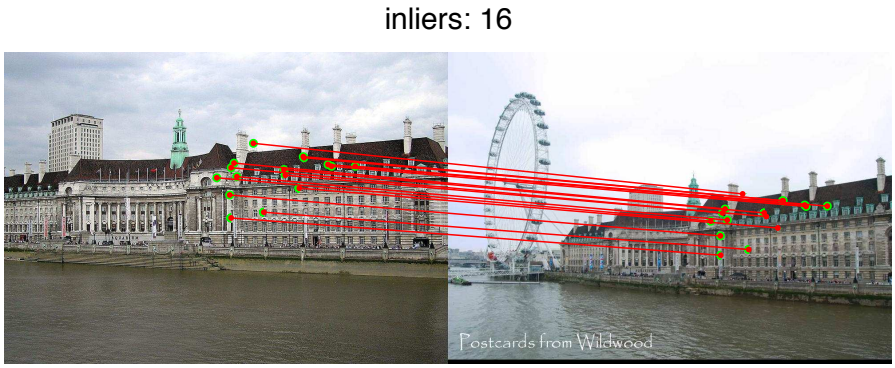
As with NOSAC, this is done for every correspondence and the inliers from the best estimation used to compute a final estimate for the transformation. The process is faster than RANSAC and also results in a 24% increase in yield compared to the baseline system's yield, giving an absolute yield of 22.5%.

5.3 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 [4] notes that using a different distance measure



(a) Affine Invariant SIFT



(b) Affine Invariant RootSIFT

Figure 5.3: The difference in matching by using RootSIFT descriptors as compared vanilla SIFT.

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 5.2 shows the conversion of the SIFT descriptor `sift` to RootSIFT space.

Listing 5.2: 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, giving an absolute yield of 24.5%

5.4 Results

The results of these geometric improvements is shown in Table 5.1.

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 5.1 shows the impact on yield (see Subsection 3.3.2) by implementing these improvements and varying the inlier threshold.

It is easy to see that implementing the affine invariant detector with the RootSIFT mapping results in the best performance. This is due to the increase in matching accuracy, allowing a lower inlier threshold to be used and so more matches to be made.

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

Chapter 6

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 yield performance of the system.

6.1 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 test 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 made.

Figure 6.1: A selection of images from Wikipedia for the object `10_Downing_Street`.

6.2 Database Augmentation

Existing work addresses this problem by including a process known as database augmentation, the background of which is given in Chapter 2.

The database-side feature augmentation 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 ground-truth classes of each image are already known, and it is assumed that overlap between classes does not exist[1]. In the 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 are non-existent.

6.3 Turbo-boosting

While database-side feature augmentation is not suited to this application, the overall idea of feature augmentation can still be used, with a process called *turbo-boosting*.

Turbo-boosting is the augmentation of features from images of a completely independent dataset. The

[1] This is a contentious point. See the footnote in Subsection 3.3.2.

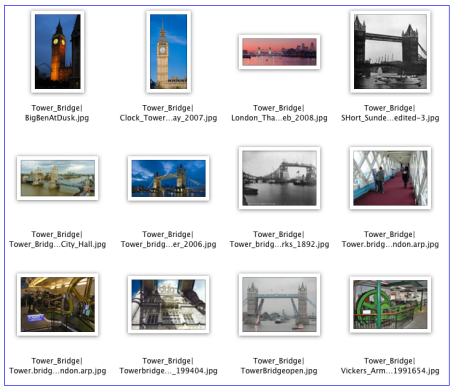
process is exactly that of spatially verified database expansion [4], however rather than the model images being used for augmentation, labelled images from Microsoft's Bing image search are used. A number of images, called *turbo images* are downloaded for each class, by issuing a query to the Bing search engine (see Section 3.2). For example, the turbo images for the class `Tower_Bridge` are shown in Figure 6.2b. For each model image, if a turbo image matches it, the visual words of the turbo image are projected onto the model image.

Take, for example, the image in Figure 6.2c as the first model image to be turbo-boosted. Each turbo image is tested to see if it matches the model image. In this example, the turbo image in Figure 6.2d matches against the model image, with the match being shown in Figure 6.2e. The matched region is the bounding rectangle of the spatially verified visual words, with a slight margin. *All* of the visual words contained with the matched region of the turbo image are projected onto the model image, as in Figure 6.2f. The model image has now been augmented with the visual words of the turbo image.

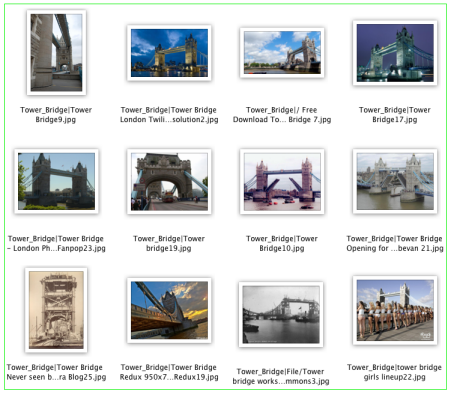
This is repeated for every turbo image, for each model image of each class. If no match can be reasonably made with a turbo image, no augmentation occurs and the process continues with trying the next turbo image.

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.

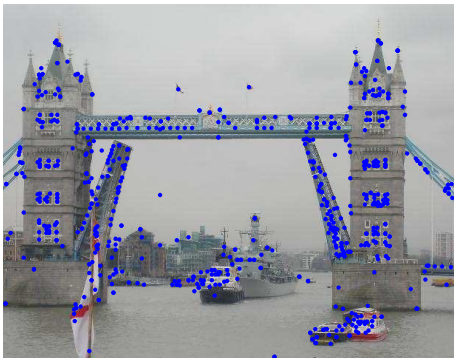
Turbo-boosting can be thought of as expanding the information content of the ground-truth image database (the model images) using an external dataset. Each model image is turbo-boosted in turn, outputting a new, more informative model image, which is used in place of the original to create a turbo-boosted dataset.



(a) Model images

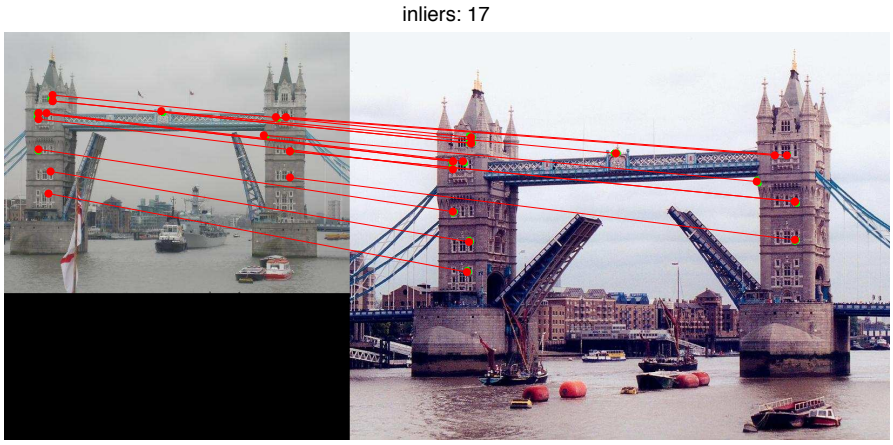


(b) Turbo images

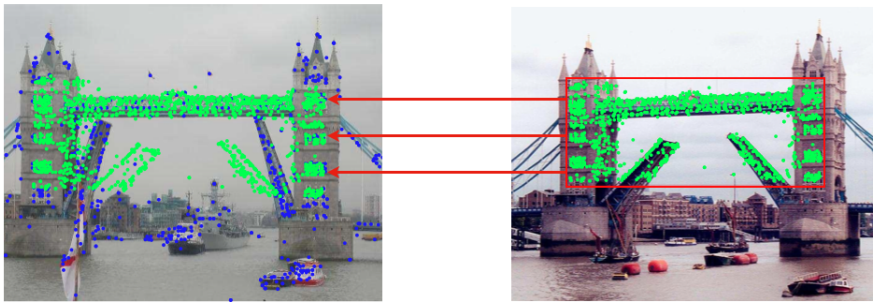


(c) Model image words

(d) Turbo image words



(e) The turbo image is spatially verified with the model image



(f) The words in the spatially verified region are used to augment the model image

Figure 6.2: The turboboosting process.

6.4 Results and Analysis

Including turbo-boosting in the pre-computation process dramatically increases the yield performance of the object recognition system. Table 6.1 shows the yield performance of the system when turbo-boosting was included. The number of turbo images that are downloaded was varied. The results (shown in Table 6.1 and Figure 6.3) show that increasing the number of turbo images downloaded increases the yield, however this effect flattens out above 25 images. This is because downloading additional turbo images will not bring in many new features to augment the model images – the model images become saturated with data.

	# Turbo images downloaded						
	0	5	10	15	25	35	50
Yield	22.6%	25.2%	26.1%	27.8%	31.1%	31.2%	31.7%

Table 6.1: The yields achieved using different levels of turbo-boosting with the Affine RootSIFT configuration and an inlier threshold of 5.

Overall, turbo-boosting increases the performance by 75% with respect to the baseline system, and 29% compared to the best possible configurations without turbo-boosting.

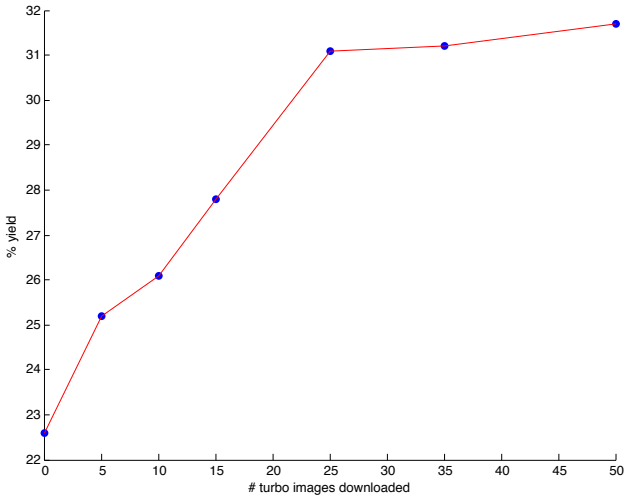


Figure 6.3: The effect on yield by increasing the number of turbo images downloaded.

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 of 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 creates a more 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.

Figure 6.4 shows a case where the query image Figure 6.4a cannot be matched to any model images. However, the model image Figure 6.4b can be turbo-boosted, leading to the augmentation of the words in the matched region of the turbo image Figure 6.4c. The turbo-boosted model image now matches against the query, as shown in Figure 6.4d. There are many examples of cases similar to this one – where the model image is perfectly representative of the object, but does not have enough stable visual words to match reliably. However, due to the large number of turbo images, some augmentation occurs which pulls in better stable visual words.

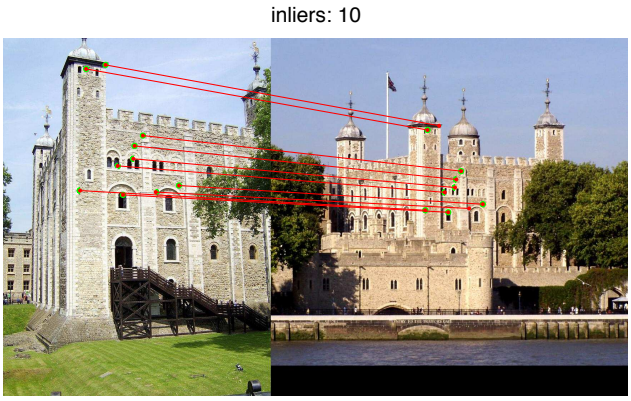
There are cases where turbo-boosting makes no difference to the recognition of an object. This is mainly due to there being no turbo images which match to the model images, so no augmentation occurs. This is an issue which is unavoidable due to low number of representative model images for some objects.

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 $O(N_c\overline{N}_m N_t)$ 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. The increase in pre-computation time can be overcome as the process is easily parallelised – the inner loop on each class is completely independent. Another disadvantage is the slight increase in storage needed. The downloaded

(a) Query image

(b) Model image

(c) Turbo image



(d) The match achieved due to turbo-boosting

Figure 6.4: The query image Figure 6.4a will not be identified unless model image Figure 6.4b is augmented with the words of Figure 6.4c.

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.

Chapter 7

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.

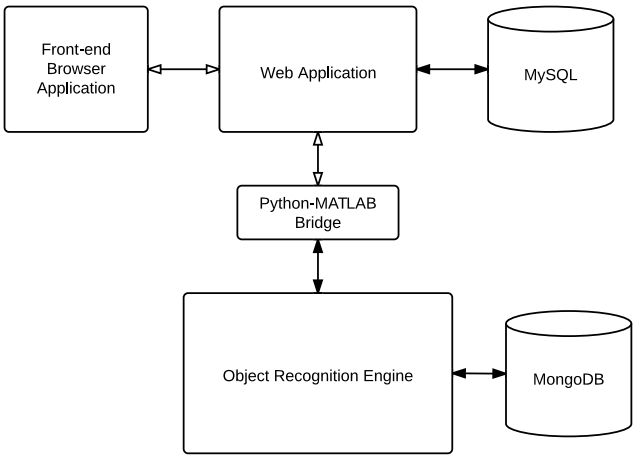


Figure 7.1: An overview of the system architecture.

Figure 7.1 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 7.1 and Section 7.2 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 7.3.

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.

7.1 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.

7.1.1 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 7.1 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 7.1: 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
        }
}
```

MongoDB is run as a daemon and the location of the database's 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

[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.

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

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 7.1 the frames file will be stored in the frames directory with the filename `4f7317501b515eaa6f148c5d-frames.mat`.

7.1.2 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.

7.1.3 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 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, each host launching a MATLAB

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

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

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

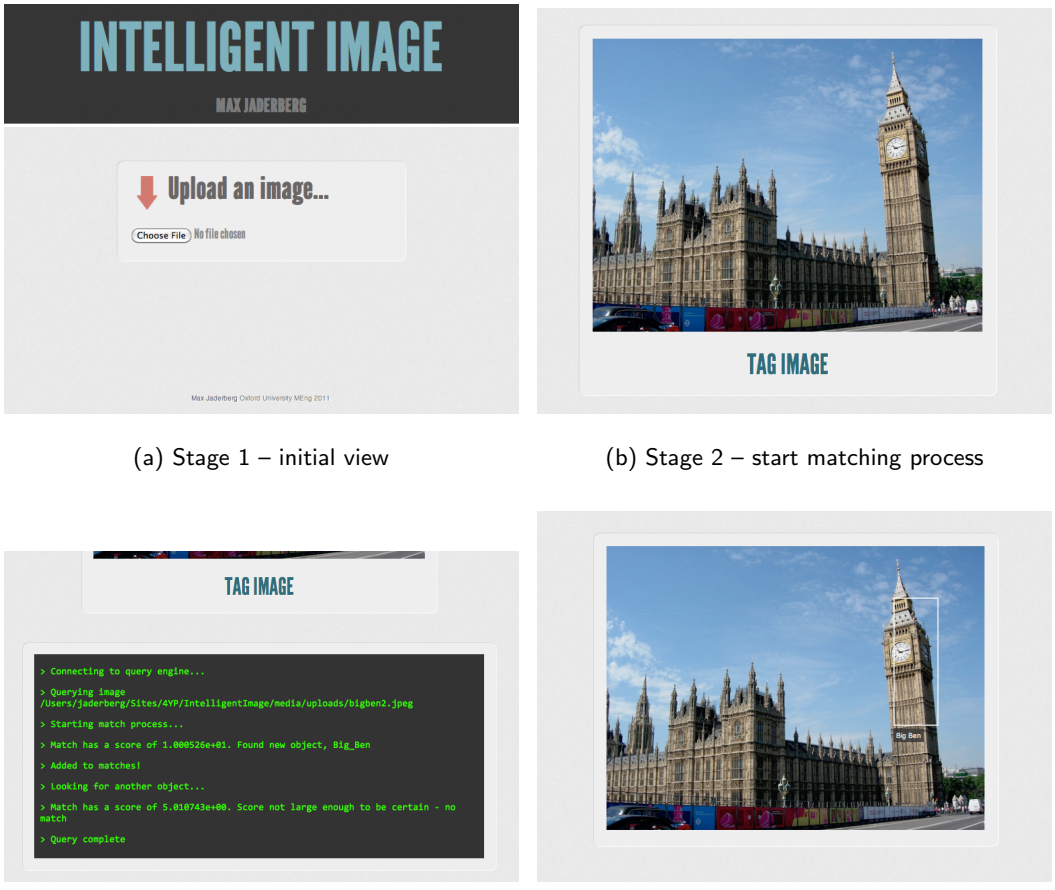
process and executing a particular function. All jobs generate a finished flag so the pre-computation script can keep track of the dispatched jobs and synchronise the process.

Using the distributed system for pre-computation and testing results in a far quicker process, reducing the time from 15-20 hours for some jobs to around 1 hour.

7.2 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.

The web application is divided in to two parts, the front-end application that is run in a user's web browser (Section 7.2.1) and the back-end web server (Section 7.2.2) 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 7.2: Screenshots showcasing the web application's functionality.

Figure 7.2 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.

7.2.1 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 7.2a). 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 7.2b).

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

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

Upon clicking the "Tag Image" button, the real-time log is displaying informing the user of the progress of the matching process (Figure 7.2c). 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 7.2) 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 7.2d). On click, the Wikipedia page for the object is opened. This is achieved purely with HTML and CSS.

Listing 7.2: 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>
```

7.2.2 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 7.3 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`.

7.3 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 developed using MATLAB's Java runtime environment. This is open source and available from [18]. 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 7.3 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 7.3: 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 function becomes trivial and so the web application can run the MATLAB matching process.

7.4 Examples

Figure 7.3 shows various example results and a video of the implemented system can be found at [21].

Figure 7.3: A selection of annotated images. Clockwise from top left: London Central Mosque, Abbey Mills Pumping Station, Bloomsbury Square, University of Greenwich, Victoria and Albert Museum, and St Jame's Church, Piccadilly.

Chapter 8

Summary

This project aimed to perform large scale image retrieval to provide automatic object recognition to images. The object classes were taken from Wikipedia, and for the purpose of this project, a subset of objects "Structures in London" was used for development and testing.

The image data for the model was crawled from Wikipedia, while the turbo-boosting images and test images were gathered from Bing and Google respectively.

A baseline system was built to incorporate all the functionality desired. The resulting MATLAB object recognition engine can recognise and tag multiple objects in a single image using the Wikipedia data, and this communicates with a Python web application to allow users to interact with the system over the internet. The performance achieved by this baseline system was a yield of 18.1%, leaving a lot of room for improvements.

A number of extensions to the baseline system were researched and implemented to improve the performance. Alternative geometric models were explored such as NOSAC, and an affine invariant feature detector was used. In conjunction with mapping the feature descriptors from SIFT space to RootSIFT space, these improvements helped increase the recognition performance by a maximum of 35% compared to the baseline system, to a yield of 24.5%.

A novel method was developed called turbo-boosting to further build upon the baseline system in an independent way to the geometric improvements. The turbo-boosting process augments the model images with images gathered from Microsoft's Bing. The turbo-boosting, combined with the geometric improvements, dramatically increases the performance of the system – recognition increased by 75% compared to the baseline system, giving a yield of 31.7%.

The results of the project show that a user friendly object tagging website can be built with a maximum yield performance achieved of 31.7%. This is good, but still leaves many objects unrecognised. The images that were not able to be recognised are generally where the image shows a very different viewpoint, or focuses on a different aspect of the object, which does not appear in the database of model images. This is to be expected, as the dataset used to create the database, the images from Wikipedia, is very noisy and sparse when it comes to representative images. Images from Wikipedia may only contain one or two images of an object, and so do not cover the range of views that, for example, an inquisitive tourist may capture. While methods like turbo-boosting help in bolstering the Wikipedia dataset, the overall lack of representative ground-truth images is thought to be the main limiting factor.

Further work would aim to increase the yield and scale the size of the database. For the yield, better filtering of the Wikipedia dataset could be done, as well as the removal of distracting features that do not uniquely define a class [2]. The textual content of Wikipedia articles could also be harnessed, together with the articles' images, to find more ground-truth images to represent objects. Also a general class representation of the model images could be created, so that initial matching is done on a class level, and subsequently on an image level, thus reducing the amount of in memory data required. The labelling of objects could also be done in a more probabilistic way – instead of simply defining the object found by the class of the first image that is spatially verified, use some algorithm to determine the most likely object given the top spatially verified images, with some uncertainty score.

Bibliography

[1] T. Quack, L. Bossard, S. Gammeter and L. Van Gool. I know what you did last summer: object-level auto-annotation of holiday snaps. In *ICCV*, 2009.

[2] 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.

[3] C. Silpa-Anan and R. Hartley. Localization using an image-map. In *Proc. ACRA*, 2004.

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

[5] Y. Aashcim, M.Lida, and K. Risvik. Multi-tier architecture for web search engines. In *Proc. Web Congress*, 2003.

[6] L. Barroso, J. Dean and U. Holzic. Web search for a planet: The google cluster architecture. In *Micro, IEEE, 23*, 2003.

[7] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *Proc. CVPR*, 2006.

[8] Y. Amit and D. Geman. Shape quantization and recognition with randomized trees. In *Neural Computing, 9(7):15451588*, 1997.

[9] J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *Proc. ICCV*, 2003.

[10] D. Lowe. Object recognition from local scale-invariant features. In *Proc. CVPR*, 1999.

[11] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proc. CVPR*, 2007.

[12] K. Mikolajczyk, B. Leibe, and B. Schincle. Multiple object class detection with a generative model. In *Proc. CVPR*, 2006.

[13] V. Lepetit, P. Lagger, and P.Fua. Randomized trees for realtime keypoint recognition. In *Proc. CVPR*, 2005.

[14] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University PRess, ISBN: 0521540518, second edition, 2004.

[15] A. Bosch, A. Zisserman and X. Munoz. Scene classification via pLSA. In *European Conference on Computer Vision*, 2006.

[16] J. Knopp, J. Sivic and T. Pajdla. Avoiding confusing features in place recognition. In *European Conference on Computer Vision*, 2010.

[17] T. Quack, B. Leibe and L. Van Gool. World-scale Mining of Objects and Events from Community Photo Collections. In *CIVR*, 2008.

[18] `https://github.com/jaderberg/python-matlab-bridge`

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

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

[21] `http://www.youtube.com/watch?v=NFFOjgw7QhM`

[22] `https://www.dropbox.com/s/lbdcjd58bs1rmot/List%20of%20structures%20in%20London.pdf`