Week 3 & 4: k-means & Bag-of-Visual Words or how to create a Google Goggles-like system

Efstratios Gavves *

November 18, 2013

1 Summary

1.1 Provided:

- Handout
- Basic code and functionality
- Dataset

1.2 Requested:

- A report. **The questions which you need to answer in the report will be marked with red font in this handout.** The report should be delivered at `efstratios.gavves@gmail.com`. The **deadline** for delivering report 2 is Friday 29th of November, 23:59:59. Delay in delivering the report will be penalized with **minus 1 point per day**.
- A zip file with the implemented code. The places where you are suggested to put your implemented code (this time in *week3.py*) will be marked with comments

  `[# WRITE YOUR CODE HERE], or something similar.`

  You **should not** write code where there is a comment

  `[# DO NOT TOUCH]`

  We do so to ensure that the final code will look as clean as possible. That way we will use our time in evaluating your code and giving you feedback, and **not debugging it** or trying to understand it. We are not responsible for debugging, please make sure the code you send us works. If there are bugs you will be asked to re-submit once **at most**.

*Email: `efstratios.gavves@gmail.com`, Room: C. 3.244, Office hours: Mon 17.00-18.00

1.3 Useful links:

- `https://www.enthought.com/` Canopy Enthought environment
- `http://effbot.org/imagingbook/pil-index.htm` Python Imaging Library Handbook
- `http://docs.scipy.org/doc/` Python Numpy and Scipy Library documentations
- `https://piazza.com/uva.nl/fall2013/uva/home` Piazza.com, the electronic embodiment of our class, where you can ask questions and we will answer. Asking questions here is good, as everybody can profit from the answer.

2 Preliminaries

2.1 Summary

In the third week's assignment we will get more familiar with the one of the most popular, if not the most popular, clustering algorithms, that is k-means . We will first work on our own, not-optimized version of k-means , so that to get a better grasp of what it is doing, and then we will proceed with the one provided by the `scipy` package. After having understood how k-means works, we will show some nice applications for it, first on image segmentation and second on image retrieval (similar to week 1). Remember, similar methods can easily applied in the textual or the music domain. For example, we will use SIFT to describe images, for music we could use the so called MFCC features. Or in text we would use the actual (stemmed) words.

- Understand k-means
- Show how to use k-means for color based image segmentation
- Show how to use k-means to create a Bag-of-Words system
- Create an actual Bag-of-Words system

2.2 Download the provided package for Week 3 & 4

First download the assignment package from

`http://staff.science.uva.nl/~gavves/files/multi_week3_4.zip`.

Extract the package in your working directory $main_dir, like in the previous week e.g.,

`C:\Users\username\multimedia_course`

Before proceeding please **make a backup** of your previous code. Taking a backup is essential, better be safe than sorry.

3 Understanding k-means

k-means is a quite simple algorithm, which clusters nearby items together. First, generate randomly some data,

```
x, labels = week3.generate_2d_data()
```

This function will returns an array `x` which contains the two dimensional data and `labels` and their labels. You can plot them with their labels

```
week3.plot_2d_data(x, labels, None, None)
```

and you should get something like Fig. 1. The `plot_2d_data` function takes as input 4 arguments. The first one is the data that will be plotted. The second one is the labels of the data, according to which the data will be colored. If you want to give them no particular colors, you put the second argument to `None`. The third argument is to which cluster each point is predicted to belong. The forth argument is the location of the centers of the clusters. In our example we have three labels, so each point is colored according to its label.

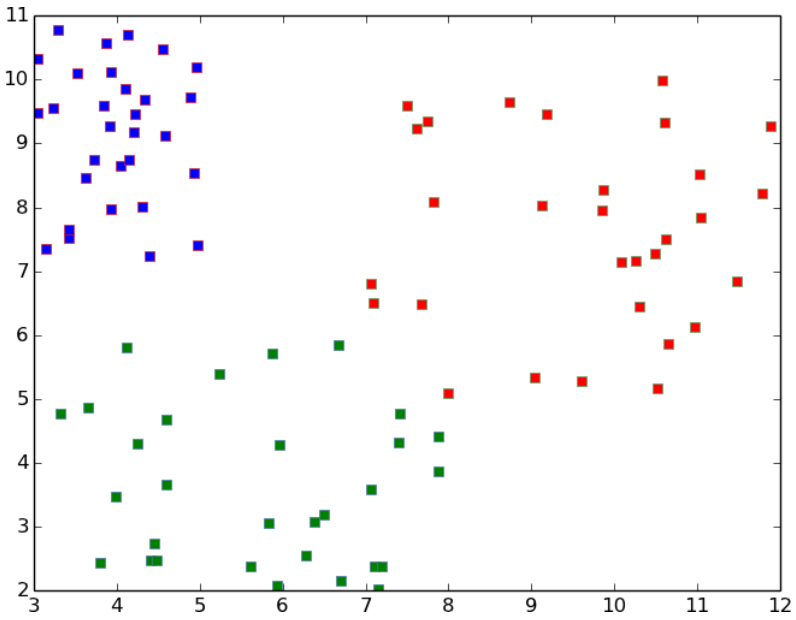

Figure 1: Random 2D data.

Step 0. As a preliminary step, we need to decide how many clusters we want in the end. Also, we want some initial positions for these clusters. Therefore, you now need to set the parameter `K` that defines the number of clusters and pick at random `K` points from x as centers

```
K=2
means = np.array(random.sample(x, K))
```

You now can plot the data and your initial centers using the function `plot_2d_data` from `week3`. The result, without coloring according to labels, should look something like Fig. 2. The colored cross symbols denote the current location of the

centers. ***Note here*** *that the centers are supposed to be picked at random, so you might have different pictures that in the ones in the handout. Furthermore, the final results will be very influenced by the initialization of the centers. So, if you re-run the code, you will probably get different results every time.*

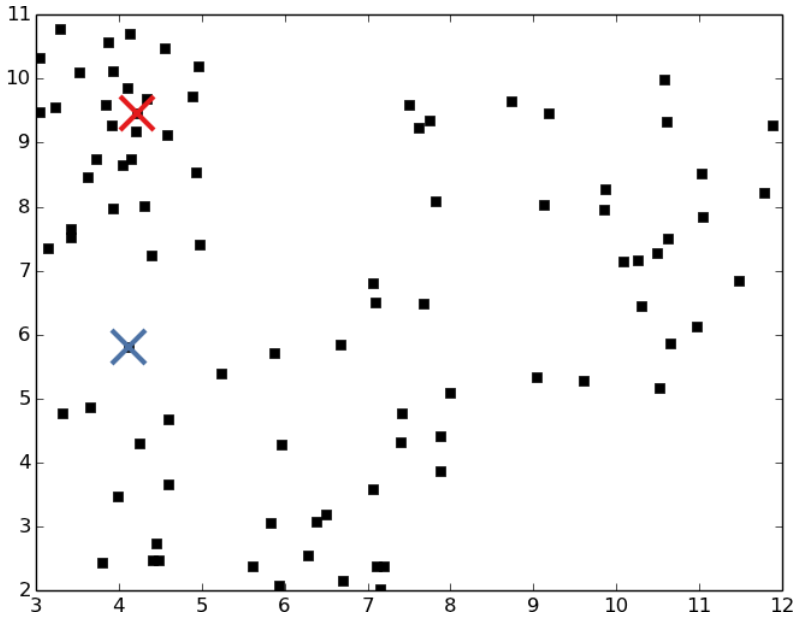

Figure 2: Random means on the 2D data.

Step 1. Having already some centers, we need to compute the **Euclidean** distances between the centers and the data `x`. You can use the code that you programmed in the previous assignments.

```
dist = np.zeros([K, x.shape[0]])
for i in np.arange(0, K):
    for j in np.arange(0, x.shape[0]):
        ...
```

Step 2. Now that we have the distances between the centers and the data, we need to find to which center each point is closest to. Since we have Euclidean distance, smaller values indicate that points are closer. Therefore, we can use the function `np.argmin` to find the closest points

```
closest = np.argmin(...
```

Since we know to which center each point is the closest to, we can visualize them accordingly

```
week3.plot_2d_data(x, None, closest, means)
```

If you did everything correctly, you should get something like Fig. 3. The assignments are marked with the colored circles surrounding the points.
Step 3. So far, we have already some centers and we also know to which center each point is assigned to. It is therefore easy to re-calculate the location of the

center $center_i$ for the cluster C_i, **using only** the points that are assigned to this cluster

$$center_i = \frac{1}{C_i} \sum_{x_j \ in \ C_i} x_j \tag{1}$$

If you plot now the points with the new centers (and the old assignments), you will get something like in Fig. 4

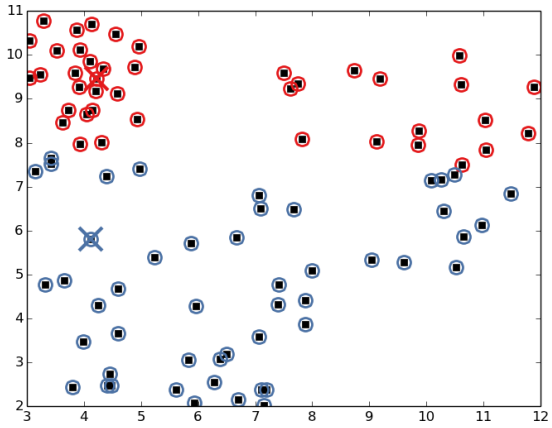

Figure 3: To which center each point belongs to.

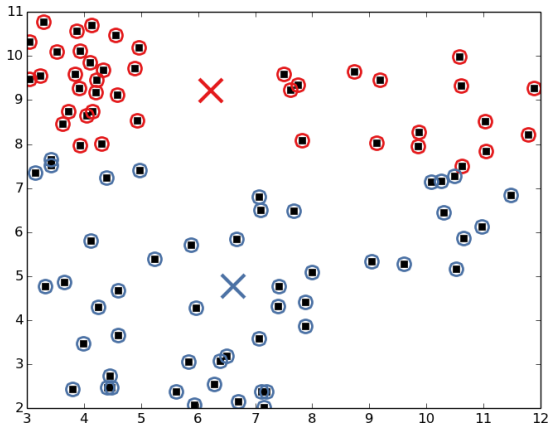

Figure 4: Re-calculating the centers.

In the above we made use of K=2. Repeat the steps 0-3 for K=3, 5, 10. Generate figures like 2, 3, 4 and paste them in your report. In the end you should paste 9 figures. Report your observations. For example what is the best K according to your opinion and why (here you could also color the points according to their true label)? What happens when you set K too large or any other interesting observation you might think of? What is the result when you re-run the algorithm with differently (randomly picked) initial centers?

CONGRATULATIONS. You just finished one round of the famous k-means algorithm. To summarize, after initialization (i.e. defining K and initial centers) the k-means algorihm is composed of three sequential steps, that is *i)* compute distances between centers and points, *ii)* find out to what center each point belongs to and *iii)* re-calculate the centers, based only on the points that are assigned to that center. Then, the k-means repeats these three steps for several iterations, until convergence.

Paste your code into the function `week3.mykmeans`, which already contains the for loop for repeating the steps. Run now `week3.mykmeans` on your data above for 20 iterations. Plot figures like 4 for iteration 1, 5, 10, and 20. Namely, in the end you should paste 4 figures. Report your observations.

The k-means algorithm might end after a predefined number of iterations. Based on the lectures, can you propose other criteria that would signal the algorithm to stop, without needing to wait until all iterations are complete?

4 k-means & Image segmentation

Now that we have a good grasp of what k-means is, and what it does, we can actually use it for things. From now on we will use the `kmeans` function provided from the `scipy` library. A nice first application is on color-based image segmentation. This segmentation and compression is quite similar to the **.gif** images we know, usually from funny cat pictures in the web (`http://www.pbh2.com/humor/funniest-cat-gifs/`).

Please, open the corals image, Fig. 5 and reshape it to get an array of all the (R,G,B) triplets

```
im = Image.open('../../data/coral.jpg')
imshow(im)
im = np.array(im)
im_flat = np.reshape(im, [im.shape[0] * im.shape[1], im.shape[2]])
```

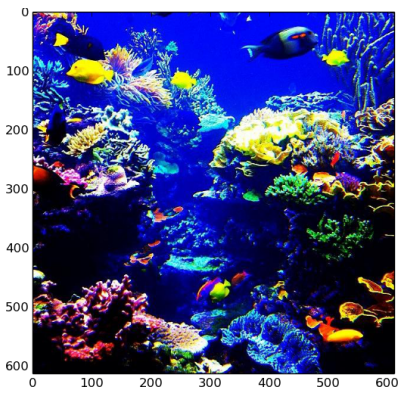

Figure 5: Corals.

Each pixel will be one of this (R,G,B) triplets and we want to run the k-means algorithm on all these points to cluster the colors. Since the number of

pixels can be quite large and thus k-means could be quite slow, we randomly pick some of them to use. We do so by the `random.sample` command.

```
im_flat_random = np.array(random.sample(im_flat, N)
```

As a rule of thumb, the number of points N we need to run k-means should be at least $N \geq 10K$. For example, if we want 100 centers, we should have at least 1000 points. Bear in mind though, more points give always more accurate results. For example, I used 10,000 points for all cases in this example of image segmentation. If possible you should keep as many points as your computer and your patience can tolerate.

We define the number of different colors we want our final picture to have and run the k-means algorithm

```
K = 10
[codebook, dummy] = cluster.kmeans(...
```

After the algorithm has ended, we have our codebook of colors, that is the color centres based on which you can describe all other colors in your image. Although it is quite cool that we found some center colors, it would be even cooler to actually use them for something. So, now visit every pixel in the image and assign it to the closest color center. One way to do that would be to use the `argmin` function as before. However, you could also use the vector quantization command, that is `cluster.vq`, that does exactly the same thing, probably faster though. Having found to which color each pixel is assigned to, you can generate your segmented/compressed/quantized image, as in Fig. 6 (K=10).

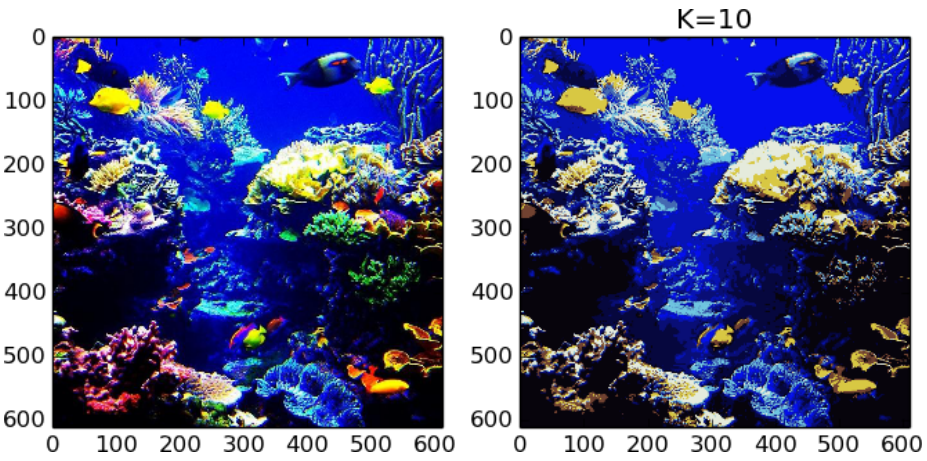

Figure 6: Corals.

Repeat the image segmentation for K=3, 5, 10, 50, 100. Paste pictures similar to the right picture in Fig. 6. What do you observe in the quantized pictures for very large or very small values of K?

5 k-means , SIFT & Bag-of-Words

5.1 Visualizing Bag-of-Words

A second, and quite important, application for k-means is the Bag-of-Words model. As we saw in the previous week, SIFT is a quite good, and robust feature, for describing an image, invariant to several sources of noise that usually appear. However, it is too big, composed of 128 values per vector.

Make a rough calculation (on paper or with your pocket calculator), how much storage in megabytes would we need to store the 50,000 SIFT vectors per image (this is how many SIFT vectors state-of-the-art systems use), assuming that each SIFT dimension is stored as an integer (1 integer=2bytes). Now multiply this number with the 350 million pictures uploaded on Facebook per day. How much storage would Facebook need to be able to store and process all the new images it gets **in a single day?**

Instead of explicitly storing each SIFT, however, we could use k-means to save a lot of space. More specifically, similar to what we did with the segmentation example, one needs to *i)* run the k-means algorithm, on SIFT vectors and not (R,G,B) triplets this time, *ii)* compute a k-means SIFT codebook, *iii)* vector quantize each SIFT vector and finally store only a single number instead all SIFT vectors explicitly. Naturally, there is some information lost there, but the gains are magnificent.

Since it would be rather laborious for you to run on you computer k-means on SIFT vectors, we provide you one composed of K=100 centers. You can read it using the following command

```
codebook = week3.load_codebook('../../data/codebook_100.pkl')
```

Given the codebook centers (which we will call from now on "words"), one can assign each SIFT to these words. Then, each SIFT can be visualized as a different color. Run the provided code

```
word_patches = week3.show_words_on_image(...
```

The function `show_words_on_image` takes 7 arguments as input.

- The first one is the path to the image.
- The second one is the number of clusters/words.
- The third and the forth ones are the frames and the SIFT vectors as returned from the `compute_sift` function.
- The fifth argument is the word indices for each of the SIFT vectors.
- The sixth argument is the colors according to which each word is colored.
- The last argument will be explained in the next paragraph.

If everything is ok, you should get something like Fig. 7 for the image "all_souls_000057". Each color point now corresponds to a certain word, as you can also see in the colorbar. Note that in the code we give you the colors and therefore also the colorbar are generated randomly. So, if you re-run the code for generating the colors, you will get different colors every time.

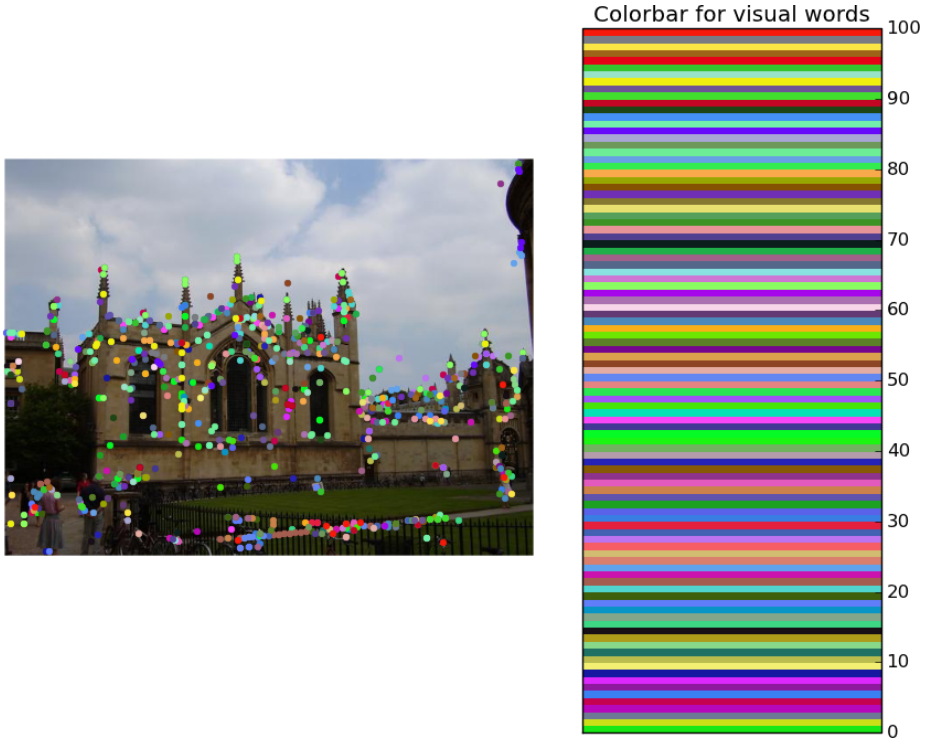

Figure 7: SIFT vectors that have been assigned to words, plotted on image.

Repeat for 5 random images from ‘‘`../../oxford_scaled`’’, plotting the visual words as we do in Fig. 7. An interesting case is when an image contains repetitive structures, like ‘‘`all_souls_000003.jpg`’’. What do you observe?

Note that the function `show_words_on_image` returns an argument called `word_patches`. These are the patches where SIFT was extracted from, without considering the different orientations though (therefore the patches could perhaps needed to also be rotated before visualized). Nonetheless, you can get a good impression of what the contents of each visual word look like. If you visualize them using the provided code, you should get for the 0-th word something like Fig. 8. The patches were collected from my 5 random images, so you should get slightly different patches.

Use the provided code to visualize the contents of 5 different visual words of your choice. What do you observe?

5.2 Bag-of-Words image representation

In the previous subsection we saw how we can transform the SIFT vectors extracted from an image as “words”. Next, we would like to aggregate the information from all the words in the image, so that to represent the image itself. *What kind of representation would make sense, though?*

To answer this question, we get inspired from week 1, and attempt to express the image as a histogram. Different from week 1, however, where an image was

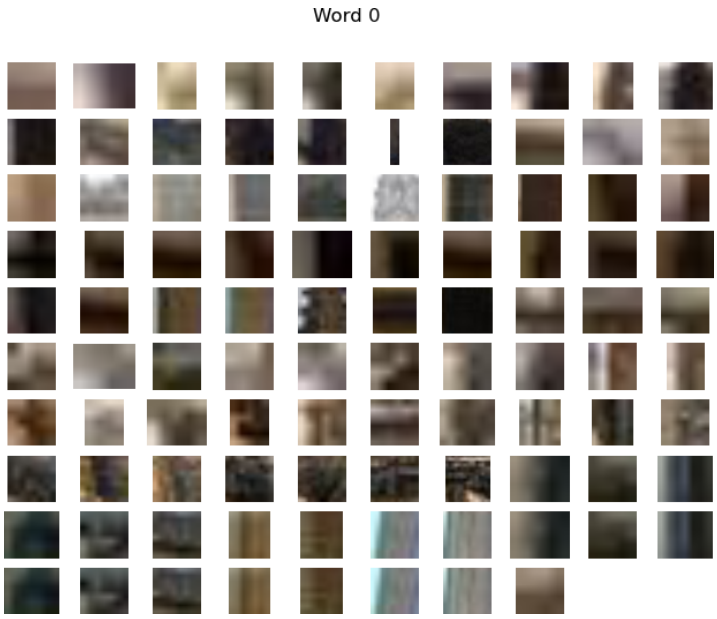

Figure 8: Patches for word 0.

represented as a histogram of R, G, B colors, now we create a "word" histogram, where each bin tells us how many times a word occurred in the image. Hence, similar to week1, we can use the `bincount` function to compute the Bag-of-Words histograms.

```
bow = np.bincount(...
```

Extract the Bag-of-Words image representations for the 5 images you used previously. Visualize the histograms using the function `matplotlib.pyplot.bar` that you used in week 1 and paste the results to your report.

6 Bag-of-Words & Image Retrieval

6.1 Retrieving images

Next, we will use the Bag-of-Words image representations to perform query-by-example retrieval similar to what we did in week 1 and week 2. To make it easier for you, we have pre-computed the Bag-of-Words vectors per image for different codebook sizes and stored it in the directories `''../../data/bow/codebook_10''`, `''../../data/bow/codebook_100''`, etc. You can read a Bag-of-Words representation using the following commands

```
f = 'all_souls_000026.jpg'
bow = week3.load_bow('../../data/bow/codebook_100/' + f + '.pkl')
```

Now that we have the Bag-of-Words vectors for all our images, we can perform query-by example retrieval and find the most similar images to a specific

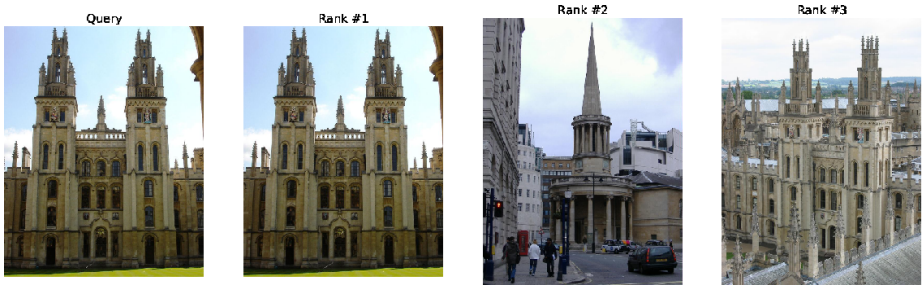

Figure 9: Retrieved results using Bag-of-Words with a codebook K=100.

image. For the ℓ_2 distance, ℓ_2 normalization, K=100 and `'all_souls_000026.jpg'` as a query, the result should be something like Fig. 9

Start from the query images "all_souls_000065.jpg", "radcliffe_camera_000390.jpg" and "christ_church_000190.jpg" in the "../../data/oxford_scaled/" directory. Given these queries find the most similar pictures in the "../../data/oxford_scaled/". Use the ℓ_2 and the histogram intersection distances and experiment with the appropriate normalization techniques, as we described them in week 2. Find the closest matches for each of the three queries. Paste the top 5 result per query per distance type. In the end, you should paste 30 images (you can save them smaller to save space, so that in the end all of them to fit in one page).

6.2 Evaluation

A very important concept is the evaluation; we need to be able to tell how well our method performs under different settings, so that to be able to optimize its performance. To do so we can use *precision@N*.

The precision@N is measured as

$$prec(N) = \frac{1}{N}\sum_{j=1}^{N}[label_j == label_{query}], \tag{2}$$

where $label_j$ is the label of the j-th image and $label_{query}$ is the label of the query image. The $[label_j == label_{query}]$ is the indicator function, which returns 1, when the label of image i is the same as the label of the query, and 0 otherwise. The precision@N practically counts how many of the top N ranked image have the same label as the query image.

The precision@N is already implemented in `week3.prec_at_N` and you can call it as

```
prec = week3.precision_at_N(query_id, gt_labels, ranking, 5)
```

Repeat the experiment from the previous subsection(retrieving images similar to the Oxford queries), measuring now the precision@5 and precision@10. For the experiment use the already provided codebooks of different sizes, namely 10, 50, 100, 500, 1000 words. Create plots where you paste your results for different sizes. For example, one plot will contain on the y axis the precision@5 for the three queries for ℓ_2 distance and for their average, and on the x axis the

different codebook sizes from smaller to larger. For two distance measures and the two evaluation methods(@5 and @10), you should paste in total 4 plots. What do you observe that happens when you change the codebook size? What distance measure works better for you? Can you propose ways to improve the accuracy of the system?

Furthermore, repeat the experiment using color histograms as representation for the images, using your code from week 1 and the distance measure of your choice. Compute the precision@5 and precision@10 scores for the same queries and compare them with the Bag-of-Words representations. What do you observe?

7 Bonus 1

As part of the first bonus question, we ask you to implement your own Bag-of-Words Image Retrieval system. We expect that you compute yourself **larger** codebooks. We also expect from you **to implement an inverted index filesystem**. Last, the final retrieval should use geometry to rerank the images and keep only those that have **good enough geometrical agreement with the query**. After you implement the pipeline, we expect to rerun the last experiment and report your observations regarding the accuracy, the computational efficiency and the storage.

Obviously, this first bonus question has easy and difficult aspects. The easy part is that most of the code you are going to need is already there. You can use all the code we used so far to compute codebooks and bag-of-words vectors per image. You can also use the code for computing geometric agreement from week 2. However, you need to organize the code nicely, to be able to perform retrieval reasonably fast. Also, you need to program a little bit more, for implementing the inverted index filesystem. For the programming of the inverted index filesystem we will provide further instructions and help once we find that there are students interested.

8 Bonus 2

The second bonus is an extension of the first one and will transform your system to something very, very close to Google Goggles. In case there are students interested to proceed, we will provide further instructions on how to improve the accuracy and the efficiency of your system, using even better features, more sophisicated codebooks and a faster geometrical comparison methodology. As this step is quite advanced, we will help you quite a lot. People that are interested should contact the lab assistants for further instructions.