Part C Project Notes

Catherine Vlasov

November 1, 2018

Contents

Chapter 1

Task Documentation

All timings mentioned here are approximate. The specific results can be found in Section 3.2.

1.1 Image Curation

1.1.1 Initial Image Selection

The first step was selecting which images to use for the experiments. Flickr released a massive database of millions of images and we will use those taken by one user, referred to as `actor00003`. There are $13,349$ images taken by this user and they are on the server under `/array/vlasov/actor00003`. The largest image size in this directory is 3072×2304 pixels and information about all the images is in a file called `metadata.txt` in the same directory.

I wrote a script called `initial_curation.py` to do the initial image filtering. The script uses the metadata file to identify the images that are 3072×2304 pixels, makes all of these images grayscale, rotates the portrait ones to landscape, and places the resulting images in a new subdirectory called `size3072`. The script took around 10 minutes to run and 9539 grayscale, 3072×2304 pixel, landscape images were produced.

1.1.2 Choosing Image Sizes

The largest size is 3072×2304 since that is the largest size we have from `actor00003` and the smallest size was somewhat arbitrarily chosen to be 360×240. In the graphs with my experiment results, I will want the image sizes (specifically the total number of pixels) to be evenly distributed along the x-axis. In order to achieve this, I picked sizes such that the difference between the number of pixels in consecutive image sizes is roughly the same. I calculated this interval using:

$$\frac{3072 \cdot 2304 - 320 \cdot 240}{9} \approx 777,899 \text{ pixels}$$

It is straightforward to compute the total number of pixels in the n^{th} image size (where 320×240 is the 1^{st} size and 3072×2304 is the 10^{th} size):

$$320 \cdot 240 + (n-1) \cdot 777,899$$

Given the desired number of pixels (call it P), we can find dimensions with a 4:3 ratio that produce approximately P pixels. We do so by solving the following equation for x and then computing $4x$ and $3x$ to get the dimensions:

$$P \approx 4x \cdot 3x = 12x^2$$

The results of these computations are:

Width	Height	Total pixels
3072	2304	7,077,888
2912	2184	6,359,808
2720	2040	5,548,800
2528	1896	4,793,088
2304	1728	3,981,312
2048	1536	3,145,728
1792	1344	2,408,448
1472	1104	1,625,088
1056	792	836,352
320	240	76,800

1.1.3 Cropping

Chapter 2

Meeting Notes

2.1 07/11/18

- What I did:
  - Created the file structure on the server.
    - Each `actor00003/sizeXXXX/` directory only has one subdirectory called `cover` since we need to decide/calculate how many bits of payload to embed for each size.
  - Changed `initial_curation.py` so that constants in the file are instead passed in as command-line arguments.
    - Reran it and it was almost twice as fast (11 vs 19 minutes).
  - Finished `compute_probabilities.py`. It finds a λ such that:

    $\sum_{i=1}^{N} H_2(\pi_i) \in [m, m+1)$, where m is the payload size to simulate
  - Wrote `crop.py` and ran it for the other nine image sizes.
    - It crops 8×8 blocks evenly from the top/bottom and right/left.
    - The process is documented in Section 1.1.3.

2.2 31/10/18

- What I did:
  - Put together this document
  - Organized all documents and scripts in my (private) GitHub repository
  - Fixed `initial_curation.py` (the problem is documented in Section 3.3) and I successfully ran it on the server

    - Original images: `/array/vlasov/actor00003/original`
    - All 3072×2304, grayscale, landscape images are in a new directory `/array/vlasov/actor00003/size3072`
  - Learned how to use `pyplot`, plotted H_2 as an exercise
  - Computed the image sizes we'll use (the process and results are described in Section 1.1.2)
    - The method discussed on 24/10/18 doesn't work. It does produce equally sized intervals (in terms of the difference in the total number of pixels between consecutive sizes), but only between 320×240 and the ninth-largest size since this interval is only around $70,000$. The ninth-largest size would be 960×720, which is clearly much smaller than 3072×2304.
    - In order to get sizes linearly distributed in terms of the total number of pixels, the interval needs to be closer to $700,000$ pixels.
  - Started working on `compute_probabilities.py`

- *Is the value of λ bounded? How should the binary search (in the context of PLS) work?*
  - $\lambda = 0$ corresponds to maximum entropy (aka. maximum payload) because then $\pi_i = \frac{1}{1+e^{\lambda c_i}} = \frac{1}{2}$
  - As $\lambda \to \infty$, $\pi_i \to 0$
  - The order of magnitude of λ depends on the order of magnitude of the costs.
  - The binary search will have two stages:
    1. Exponential search to find an upper bound on λ. This will involve trying exponentially large values such as $0, 1, 10, 100, ...$ until a value is found such that $\sum_{i=1}^{N} H_2(\pi_i) < M$
    2. Suppose the first value where this inequality holds is $\lambda = 10^n$. We now do a binary search for λ with a lower bound of 10^{n-1} and an upper bound of 10^n and we want to find a value such that $\sum_{i=1}^{N} H_2(\pi_i) \in [m, m+1)$, where m is the number of payload bits.

- In Dr. Ker's paper "On the Relationship Between Embedding Costs and Steganographic Capacity" from June 2018, he writes about how if the detector knows the costs $c_1, c_2, ...c_N$, then the objective that should be minimized is $\sum_{i=1}^{N} c_i \pi_i^2$, which is the same as the objective in PLS except with the π_i terms squared.

  - This is a possible project extension.
  - The tricky part is computing the probabilities since the optimal solution is no longer $\pi_i = \frac{1}{1+e^{\lambda c_i}}$. Instead, it's $\frac{\pi_i}{H2'(\pi_i)} = \lambda c_i$.
  - The probabilities can be computed by running Newton-Raphson several times (Dr. Ker did it 8 times)
  - I don't need to tackle this now, but it's worth keeping in mind.

- When I use Dr. Ker's J-UNIWARD hack, I need to make sure that I work out the order in which the costs are written to the file.
  - It's hard to tell just by looking at the costs whether or not they're in the right order. If I'm wrong, I'll probably find out since the embedding will be very detectable.
  - It's very likely that the 8×8 blocks are analyzed from left to right, top to bottom. However, within each block the costs could be left to right, top to bottom **or** in the zigzag order used to store the quantized coefficients. I need to check this.

- Once I compute the probabilities, it might be a good idea to use Python's `random.seed(..)` method (with the image number as the seed) in order to do the embedding. It can be used to determine whether or not to change each coefficient and so I'll always get the same embedding with the same cover, modulo rounding.

- Dr. Ker has a faster version of JRM for feature extraction.

- Tips:
  - After embedding, open the stego image to make sure nothing got messed up (e.g. due to the order of the costs or coefficients).
  - It would be a good idea to write some scripts to check things like:
    * The number of coefficients that differ between the cover and stego images is $\approx \sum_{i=1}^{N} \pi_i$
    * Coefficients that differ between the cover and stego images only differ by ± 1
  - Test things out on small images (e.g. 64×64) to save time in case there are bugs.

2.3 24/10/18

- What I did:
  - Read Chapter 3 of the Advanced Security notes on steganography

- Wrote a script (`initial_curation.py`) to find all the largest images in the `actor00003` directory and then make them all grayscale and landscape (described in Section 1.1.1)
  - Wasn't quite working due to "`Empty input file`" error when performing multiple `jpegtran` operations

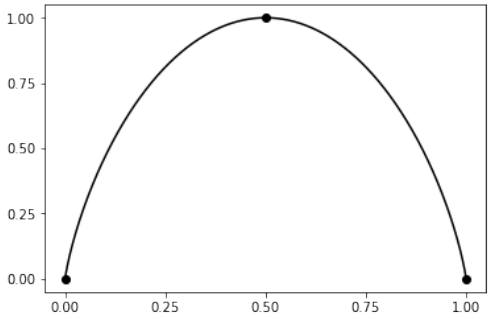
- Action plan:
  1. Calculate image sizes
     - Preserve the 4:3 aspect ratio, not because we have to but because we can and it means we can keep things as similar as possible
     - The largest image size we'll use is 3072×2304 since that's the size of the largest `actor00003` images.
     - The smallest size will be 320×240 since that's a relatively common image size (and it has a 4:3 aspect ratio)
     - The short-edge dimensions will be computed by hand by calculating $240x$ (where $x = \sqrt{1}, \sqrt{2}, ..., \sqrt{10}$) and then rounding to the nearest multiple of 24. Then the long-edge dimensions are calculated such that the 4:3 ratio is maintained.
  2. Create the directory structure on the server in `/array/vlasov/`
     - Keep a copy of all the original images in `actor00003/original`
     - Create one directory per image size, called `size3072` (for instance)
     - For each size, create two subdirectories:
       (a) One for the unaltered images, called `cover`
       (b) One per number of payload bits, called `stego-1234bits`
     - Each `cover` subdirectory will have three files per cover image:
       (a) `image12345.jpg`: the unaltered image
       (b) `image12345.costs`: the costs computed by J-UNIWARD
       (c) `image12345.fea`: the features computed by JRM
     - Each `stego-1234bits` subdirectory will have one file per stego image:
       (a) `image12345.jpg`: the stego image, which is the cover image `sizeXXXX/cover/image12345.jpg` with a 1234-bit message embedded in it
  3. Crop the 3072×2304 cover images to the sizes calculated in task 1. Do this by cropping 8×8 pixel blocks evenly from the top/bottom and right/left.
  4. Generate the costs (using Dr. Ker's slighty modified J-UNIWARD code) and features (using JRM) for all the cover images of all the different sizes.

- JRM produces 22510 real numbers (the features)
- Up to me how to store them, but ASCII is probably the most portable

5. Use J-UNIWARD to embed 0.4 bits per non-zero AC coefficient in some of the covers.
6. Write a function that takes the number of payload bits as input and computes the probabilities with which each coefficient changes during (binary) embedding.
   - Goal: given the costs $c_1, c_2, ...c_N$ (where N is the total number of coefficients) of changing each coefficient (by adding or subtracting one), compute the probabilities $\pi_1, \pi_2, ..., \pi_N$ of making each of these changes
   - Size of the payload: $\sum_{i=1}^{N} H_2(\pi_i)$
     * H_2 is the "entropy" and is defined as:

$$H_2(x) = -x \cdot \log_2 x - (1-x) \cdot \log_2(1-x)$$

     * Graph of H_2:



   - Average total cost: $\sum_{i=1}^{N} c_i \pi_i$
   - Two (equivalent) optimization problems for computing the payload size:
     (a) Distortion-limited sender (DLS)

$$\text{Maximize } \sum_{i=1}^{N} H_2(\pi_i) \text{ such that } \sum_{i=1}^{N} c_i \pi_i \leq C$$

     (b) Payload-limited sender (PLS)

$$\text{Minimize } \sum_{i=1}^{N} c_i \pi_i \text{ such that } \sum_{i=1}^{N} H_2(\pi_i) \geq M$$

- For some fixed λ, we can compute the probabilities:

$$\pi_i = \frac{1}{1 + e^{\lambda c_i}}$$

- We'll use PLS, where M is the payload size.
  - The optimal solution is when $\sum_{i=1}^{N} H_2(\pi_i) = M$
  - $\sum_{i=1}^{N} H_2(\pi_i)$ is actually monotonically decreasing, so we can find a value of λ such that $\sum_{i=1}^{N} H_2(\pi_i) = M$ for any M we choose. Then, we can compute the probabilities $\pi_1, \pi_2, ..., \pi_N$ using this value of λ.
  - The end goal is to do the embedding ourselves by modifying each coefficient with these probabilities.

- *Is 80 a standard JPEG quality factor (QF)?* In the massive image database released by Flickr, the most common QFs were 100, the QF used by iPhones, and 80. So, we're using 80 because that gives us a greater selection of images.

2.4 17/10/18

- What I did:
  - Read Chapters 1 and 2 of the Advanced Security notes on steganography
  - Read the 2008 paper "The Square Root Law of Steganographic Capacity"

- Discussed questions I had about Chapter 1 (Steganography) and Chapter 2 (Steganalysis) of the Advanced Security notes and about the 2008 paper.
  - *What is downsampling?* Shrinking
  - *When you take a pictures on your phone, what happens?* Captures raw image, immediately compresses it as a JPEG, and discards the raw image
  - *What determines a cover's "source"?* Primarily the camera. The camera's ISO setting, in particular, is very important. The subject of the photos don't make much of a difference.
  - *In JPEG compression, don't you lose some information when dividing the image into* 8×8 *pixel blocks?* No, the DCT is linear (i.e. 1-to-1 mapping from 8×8 blocks to coefficients)

  - *Is a JPEG decompressed every time you view it on a computer?* Yes
  - *When LSBR is used on RGB images, which bit(s) are changed?* Good question - it depends, but usually the LSBs of all three components (in sync)

- After embedding a payload, the original cover is destroyed. Otherwise, two nearly identical images would be floating around and Alice could easily be outed if someone got their hands on both versions.

2.5 03/10/18

- What I did: N/A
- Discussed software to be used for embedding (J-UNIWARD), feature extraction (JRM), and detection (ensemble of linear classifiers)
  - All the software is here
- Server's IP: 163.1.88.150
- Amounts of payload to embed: $O(1)$, $O(\sqrt{n})$, $O(\sqrt{n}\log n)$, $O(n)$
- $m \sim \frac{\sqrt{DC}}{2}\log\frac{C}{D}$
- TIME EVERYTHING
- I will test new embedding and new detecting methods and I could also try old embedding and new detecting methods
- Total amount of space needed (assuming around 10,000 images are used):
  - Images: $2MB \times 10000 \times 9 \approx 180GB$
  - Costs: $8B \times 5M \times 10000 \approx 400GB$
  - Features: $170KB \times 10000 \times 9 \approx 17GB$

Chapter 3

Notes to Self

3.1 Useful Commands

- Run a command in the background so that you can keep using the terminal or close it
  - `nohup python script.py &> script_output.out &`
- Check on processes that are running
  - `ps aux | grep vlasov`

3.2 Script Timings

- `initial_curation.py` (before I changed constants to command-line arguments)
  - $1131.18478608s \approx 18m51s$ (30/10/18)
- `initial_curation.py --from-dir original/ --to-dir size3072/cover/`
  - $655.185225964s \approx 10m55s$ (01/11/18)
- `crop.py --from-dir size3072/cover --to-dir sizeW/cover/ --width W --height H`, where:

W	H	Seconds	Approx. Time	Date
2912	2184	689.414359808	11m29s	01/11/18
2720	2040	662.552460909	11m02s	01/11/18
2528	1896	662.54279089	11m02s	01/11/18
2304	1728	632.872202158	10m32s	01/11/18
2048	1536	605.926501989	10m05s	01/11/18
1792	1344	555.097690105	9m15s	01/11/18
1472	1104	511.460752964	8m31s	01/11/18
1056	792	438.49830699	7m18s	01/11/18
320	240	359.436480045	5m59s	01/11/18

3.3 Lessons Learned

- The input and output file to `jpegtran` can't be the same, otherwise you get an "`Empty input file`" error.
- If it looks like directories on the server have disappeared, turn off f.lux or change the colour settings in `.bashrc`.