Assignment 2 – Arithmetic Coding

Introduction

In class, we went over a Java implementation of an arithmetic coding algorithm. The code for the algorithm has been posted in the following GitHub repository: https://github.com/kmayerpatel/COMP590Spring2019-ArithmeticCoder

Ultimately, the way in which arithmetic coding achieves compression is very similar to that of Huffman coding: symbols with high probability should take less bits to represent, while symbols with low probability should take more bits to represent. However, while Huffman coding assigns explicit codewords to each input symbol based on their expected probabilities, arithmetic coding assigns intervals on a number line to each input symbol such that the width of a symbol's interval corresponds to the symbol's probability. More probable symbols are given wider intervals on the number line, and less probable symbols are given narrower intervals.

The efficiency of arithmetic coding depends on how accurate the probability estimates are for each symbol. If a symbol's probability in the CDF closely matches its true probability in the actual source data, then arithmetic coding will produce very efficient compression. On the other hand, if a symbol's probability in the CDF is radically different than its real probability in the source data, then arithmetic coding will struggle to achieve good compression. Thus, developing an appropriate source model for the data being compressed that can accurately predict the probabilities of the symbols is a crucial aspect of arithmetic compression.

In class, we tried three different approaches for modeling source data and predicting the probabilities of input symbols. We tested these three approaches on an ascii-encoded English text file.

1. **Static probability table** – In the first method, the encoder analyzes the source data in advance to calculate the true average probabilities of the symbols across the entire input file. These probabilities are then signaled to the decoder at the beginning of the compressed file as a 4,000-byte-long header. The code using this source model can be found in the file app.StaticACEncodeTextFile.java and app.StaticACDecodeTextFile.java.

2. **Adaptive source model** – In the second method, all symbols start out with equal probability. Every time a symbol is signaled, both the encoder and the decoder update their CDFs to make that symbol more probable. The idea is that symbols that were encoded in the past are more likely to be encoded again in the future. This method does not require any header bytes for the frequency counts to be sent. However, the compression achieved is bad at the very beginning of the process, since all symbols are initially encoded with equal probability. As more and more symbols are encoded, the symbol probabilities should become more accurate and compression should improve. The code using this source model can be found in the files app.AdaptiveACEncodeTextFile.java and app.AdaptiveACDecodeTextFile.java.

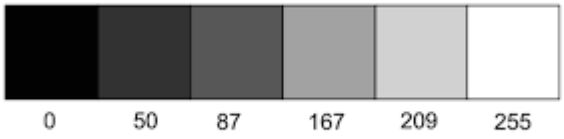
3. **Context-adaptive source model** – The final method creates an array of adaptive source models---one for every symbol. When a symbol is to be encoded, the encoder (and decoder) first selects the

appropriate adaptive source model to use for that symbol (this is called the "context"). For the implementation in class, the value of the prior symbol is what determines which adaptive source model to use. For example, if the last symbol encoded was 'm', then the next symbol should be encoded/decoded using the adaptive source model corresponding to 'm'. This method takes advantage of source data where pairs of symbols are correlated; that is, it works best when the prior symbol is particularly useful for predicting the probabilities of the next symbol. The code using this source model can be found in the files app.ContextAdaptiveACEncodeTextFile.java and ContextAdaptiveACDecodeTextFile.java.

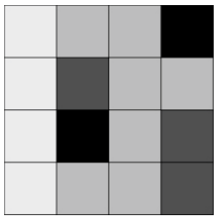
Which modeling approach is best depends on the patterns exhibited by the specific data that you wish to encode. So far in class, we've only compressed ascii-encoded English text. English text data exhibits a unique set of patterns that can be exploited. As we saw in class, the context-adaptive source model worked best for the English text we were trying to compress. This is because neighboring letters in English text are correlated with one another.

Representing Raw Video Data

For the second assignment, you will modify the arithmetic coding Java implementation from class so that it efficiently compresses a different kind of data: video pixel data. Videos are represented in the computer as a series of still-frame pictures which flash rapidly before your eyes at a rate of about 30 pictures per second. Each picture is slightly different than the picture before it, which tricks your eyes into thinking that motion is occurring. Each picture is represented in the computer as a long list of intensity values. A byte value of 255 (11111111) represents a white pixel, a byte value of 0 (00000000) represents a black pixel, and everything in between these two values represent various shades of gray. For instance, the midpoint between these two values is 128 (10000000), which represents the gray value exactly half way between black and white. Color pixel values typically require more than one byte to represent, but for this assignment we will deal with grayscale pictures only, which means each pixel's intensity value is exactly one byte long. A figure illustrating the gray color scale is shown below.



One easy way to represent a single grayscale picture is as a list of bytes corresponding to the raw intensity values of the pixels in row-major order. As a simple example, imagine a 4x4 image containing 16 different intensity values. This could be represented by a list of 16 bytes. The first four bytes represent the pixel intensities in the first row of the image, from left to right; the next four bytes represent the pixel intensities in the second row, the next four represent the third row, and the final four bytes represent the last row of pixels. This is illustrated in the figure below.



4 x 4 Pixel Image

230	170	170	0
230	65	170	170
230	0	170	65
230	170	170	65

Overlayed Pixel Values

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Intensity:	[230,	170,	170,	0,	230,	65,	170,	170,	230,	0,	170,	65,	230,	170,	170,	65]

Representing a video is easy once you understand how individual pictures are represented. A video is merely a sequence of fixed-size pictures, so the data for each picture can simply be appended to create a long series of bytes. Thus, a 30-frame grayscale video at 4x4 resolution could be represented as 30 * 16 bytes such that the first 16 bytes represent the first picture, the second 16 bytes represents the second picture, and so on.

Patterns in Video Data

Just like ascii-encoded English text, video data arranged in the way described above also exhibits patterns that can be exploited by arithmetic coding. However, the patterns in natural video data are somewhat different than the patterns we saw in English text. Here are two patterns particularly common in natural video data that can be exploited by arithmetic coding.

1. **Spatial coherence** - A pixel's intensity value is likely to be very similar to the intensity values nearby it in a single picture. Thus, pixel values change slowly as you move across a single image.
2. **Temporal coherence** - A pixel's intensity value for a frame is likely to be very similar to its value from the last frame as well as its value in the next frame. Thus, as time passes, a pixel's value likely stays the same.

These qualities can be used to make a suitable probability model for a pixel's value. Here are three ways in which this could work:

1. **Average-neighbor context-adaptive source model** – Just like the context-adaptive source model used for encoding ascii text, this model would have 256 different adaptive source models---one for each pixel intensity value. When encoding or decoding a pixel, the encoder/decoder uses the adaptive source model corresponding to the average intensity value of neighboring pixels already encoded. Note, you can not use the values of neighboring pixels not yet coded because the decoder will need to be able to replicate the exact same decision process.
2. **Prior-value context-adaptive source model** – Just like the context-adaptive source model used for encoding ascii text, this model would have 256 different adaptive source models---one for

each pixel intensity value. When encoding or decoding a pixel, the encoder/decoder uses the adaptive source model corresponding to the intensity value of the pixel in the prior frame.

3. **Differential coding** – Instead of coding raw pixel values, you could apply a preprocessing step that changes the representation of the information to reduce data redundancy. For instance, instead of coding each pixel value separately, one could store only the very first pixel value, and then represent subsequent pixel values by storing how much they differ from the prior pixel. For example, if the first four pixel values are [190, 189, 188, 189], this could be equivalently represented as [190, -1, -1, +1]. The result of this step, which is called differential coding, is a new list of numbers that are close to 0 and which may be easier to compress using one of the other source model schemes for arithmetic coding.

Assignment Description

For this assignment, your task is to implement an arithmetic source model scheme (that is, implement a version of `SourceModel<T>` and encode/decode programs that use it), that you suspect might work well for encoding video data. Ultimately, a good source model scheme is one that accurately sets the probabilities for the encoded symbols. Feel free to be as creative as you want when coming up with a source model scheme. However, it is completely acceptable to pick one (or more) of the three schemes suggested above (average-neighbor, prior-value, differential) and implement it as your own. The goal is to come up with a scheme that compresses video data well, but if your idea doesn't end up working well, that's okay too! This is all about coming up with an algorithm, implementing it, and testing it to see how well it works.

To test your scheme, I've provided a small grayscale video containing raw pixel data to compress. To keep the raw file size small, the video is only 64 x 64 pixels at 30 frames per second for 10 seconds. This means the file size is 64 * 64 * 30 * 10 = 1,228,800 bytes long. Each frame is 64 * 64 = 4,096 bytes long. I've included two versions of the video, one with a .dat file extension representing the raw data to compress, and one with a .mp4 file extension representing a version of the video that you should be able to open and watch on your computer. Please remember that you should be compressing the .dat version of the video, not the .mp4 version! The .mp4 version is only included so you can see what the video looks like that you are compressing.

Before you start designing and implementing your own scheme, try compressing the video data using the three context models we created in class which were originally intended for compressing English text (static, adaptive, and context-adaptive).

Submission Instructions

Once you've finished implementing and testing your scheme, upload your code to a GitHub repository. Submit a link to your repo along with a 1-page write-up answering the following questions about your scheme.

1. What scheme or schemes did you try? If you came up your own idea, describe it here.

2. Why do you think your scheme would do a good job predicting pixel values? How does your scheme exploit temporal and/or spatial coherence?
3. When applying the English text-based models (static, adaptive, and context-adaptive) to the video data, which scheme performed best? Does the scheme you developed compress better or worse than the English text-based models when applied to video data? If you weren't able to finish and test your own scheme, how do you think your scheme would fare in comparison to the English text-based models?
4. What is one change you could make to your scheme that might improve its results?