Compression/Linear Prediction

Do You Want to Save on Disks?

- hopefully yes!
- Compression is the art of turning some data into a new set of “data” that is on average smaller.
- Choice of lossy vs. lossless.

Shannon Entropy

- Say I have m zeros, (n-m) ones. How many ways are there to write?

- have n choose m possible orderings.

- Stirling: $n! \sim (n/e)^n$. n choose m = $(n/e)^n/((n-m)/e)(^{n-m})\ (m/e)^m)=1/(1-m/n)(^{n-m})/(m/n)^m$.

- Take $\log_2$: $\log_2()/n=-p_1\log_2(p_1)-p_0\log_2(p_0)$ where p1=(n-m)/n and p0=m/n.

- Extension to many variables: $\log_2()/n=-\sum p_i\log_2(p_i)$.

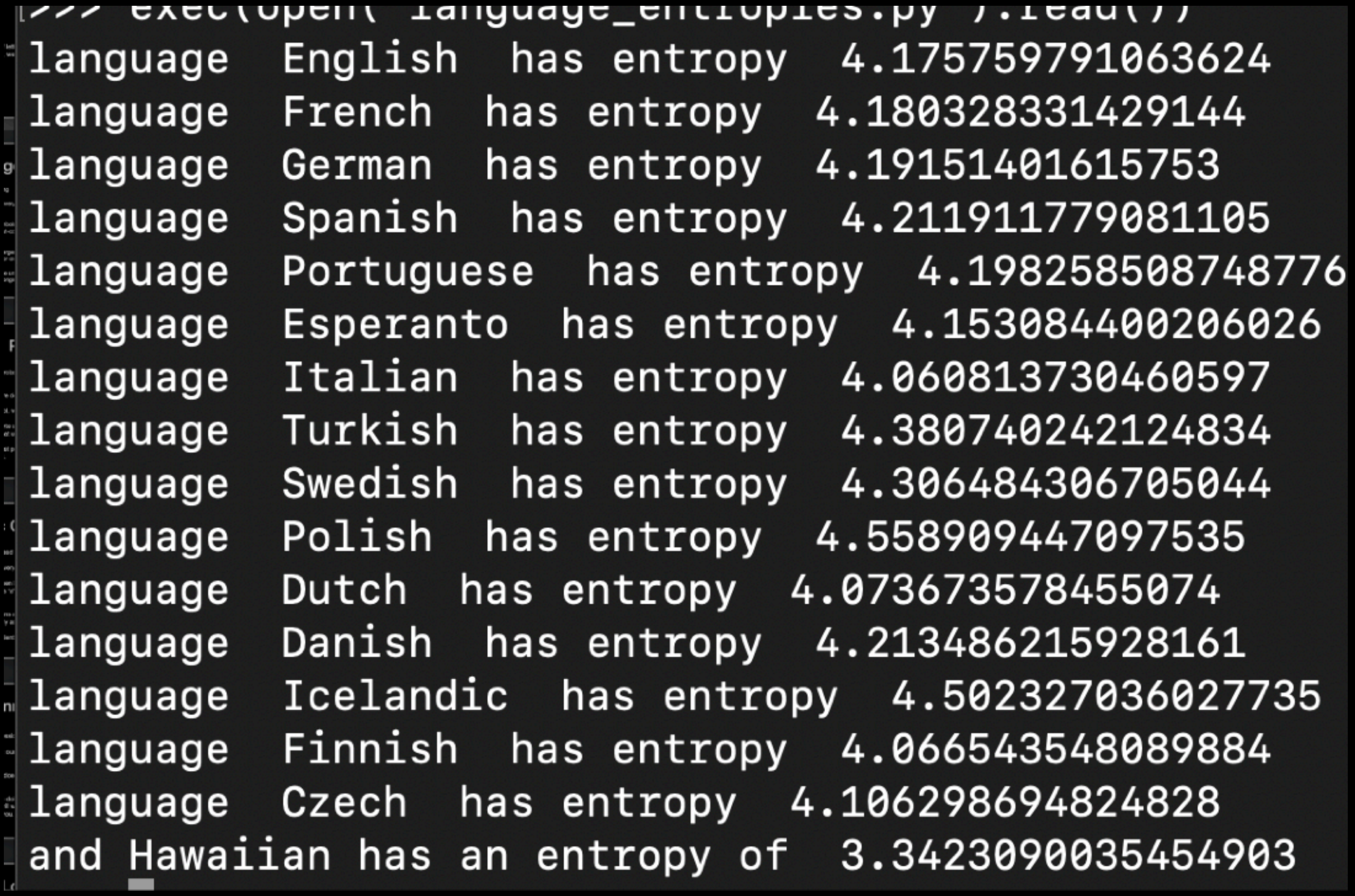
Use in Compression

- The $\log_2$ of the number of orderings is just the original number times that sum, the Shannon Entropy (s).
- I can order all the possible strings with those probabilities, then tell you which one with a binary number of length ns.
- If entropy is small, ns will be shorter than original bit stream, and I can compress my data.
- Compression is lossless - I get back exactly the bits I started with.
- Example: 10% 1, 90%0: s=-0.1*log2(0.1)-0.9*log2(0.9)=.47, so I should be able to save just over factor of 2 vs. original.

Entropy of Alphabets

- We can take wikipedia table of letter frequencies. If letters were randomly ordered, we'd expect to be able to compress.

- Of course, order is not random, so we can do much better. Shannon estimated 1 bit/letter.



```
>>> exec(open('language_entropies.py').read())
language  English  has entropy  4.175759791063624
language  French  has entropy  4.180328331429144
language  German  has entropy  4.19151401615753
language  Spanish  has entropy  4.211911779081105
language  Portuguese  has entropy  4.198258508748776
language  Esperanto  has entropy  4.153084400206026
language  Italian  has entropy  4.060813730460597
language  Turkish  has entropy  4.380740242124834
language  Swedish  has entropy  4.306484306705044
language  Polish  has entropy  4.558909447097535
language  Dutch  has entropy  4.073673578455074
language  Danish  has entropy  4.213486215928161
language  Icelandic  has entropy  4.502327036027735
language  Finnish  has entropy  4.066543548089884
language  Czech  has entropy  4.106298694824828
and Hawaiian has an entropy of  3.342309003545490З
```

How do we get there?

- Standard example is Huffman coding
- Make table of unique bit string for every unique symbol in original message
- To do: take two least-common symbols, assign 0/1 to them. merge symbols. Take next two least-common symbols, assign 1/0.
- If this one is merged, say assign merged to 1. Then assignments are 0 (for new symbol), 10 and 11 for old ones.
- Continue building tree. At end, have unique byte string for each symbol. On average, shorter than original.

Huffman in Practice

- Make a tree based on symbol probabilities, get unique mapping.
- Include special symbol for “we’re done”.
- Write byte string for each symbol, write “we’re done” at end.
- Reading: read tree, then read byte string until it matches tree entry. Gives first symbol. Repeat until “we’re done”.
- Effectively, we round p_i to nearest power of 1/2. Will not be optimal, but usually not too bad.

Arithmetic Coding

- Is rounding of n_{bits} making you sad w/ Huffman?

- Let’s map every message to a (very high precision!) number.

- Say ‘a’ has 10% probability. Then if number is between 0 and 0.1, we know first symbol is ‘a’. Then multiply number by 10, repeat.

- End up with number that contains all information about original string, but with arbitrarily accurate entropies.

- There’s an art to doing this efficiently…

Use Canned!

- Writing good low-level compression is hard!
- Our main goal will be to recast our data so that previous technique(s) will work well.
- Gzip2 does OK, bzip2 usually does better (but slower). Both have Huffman as part.
- If you understand what they're doing, you can set up your data in a way standard tools will work. Try a few, see which ones work the best for you.

Lossy vs. Lossless

- Say we have double precision data. It is *highly* unlikely all of those bits contain useful information.
- Say data have $\sigma=1$. I can round data to nearest $1/2^{nbit}$. Say I wanted to keep 4 bits of noise, then I would round my data to nearest 1/16.
- This loses information. How much?
- Think of rounding as adding a noisy signal that brings data to nearest round value. How much noise does that add?
- Round to nearest 16th means max move is 1/32, and average move is 1/64. So, adjustment stream has $\sim 1/64^2$ times original variance.
- By rounding to nearest 1/16, we have effectively increased variance by part in 64^2. Amounts to throwing away less than 1 second per hour.
- 8 bits of noise: increased variance by $1/1024^2$, or 30 seconds per *year*. You can probably afford this…
- Generally, I can't imagine any typical physics situation where you need much more than 6 bits/number.

Modelling

- We can often win (big?!) by using previous data to guess at next data. A lot less information in that error, than in full value.

- LIGO example. Say we round data so there are 10 bits from -*p*log(*p*).

- We know neighbors are similar, so if we save difference of point with previous, then -*p*log(*p*) goes to 7.

- We've done a better job reducing entropy of our data, and as reward, should save 30% on storage space.

- Can we do better job predicting?

Linear Prediction

- Linear prediction is a powerful tool to interpolate/extrapolate based on a covariance.
- Say I have noise matrix, but am missing last data point.
- What value would I give it to minimize χ^2? $\chi^2=d^TN^{-1}d$.
- Differentiate w.r.t. last data point: $(0,0,0,\ldots 1)^TN^{-1}d = 0$.
- First vector kills everything but last row of N^{-1}.
- sum(N^{-1}[-1,:-1]*d(:n-1))+N^{-1}[-1,-1]*d[-1]=0
- Let v=N^{-1}[-1,:-1]/N^{-1}[-1,-1], then d[-1]=v^Td[:-1]

Alternate LP Derivation

- $\chi^2=(d+Am)^T N^{-1}(d+Am)$ has solution $m=-A^T N^{-1}d/A^T N^{-1}A$ for scalar *m* (note sign flip, which will be convenient).

- let d bet data, except last point is zero. Let A be zero, except 1 in last point. *d*+A*m* will be total data set.

- $A^T N^{-1} A$ will then pick out bottom-right element of N^{-1}.

- $A^T(N^{-1}d)$ will pick out bottom element of $N^{-1}d$. Bottom element of $N^{-1}d$ is bottom row of N^{-1} dotted against d.

  - last entry of d is 0, so we'll get N^{-1}[-1,:-1]@d. divide by N-1[-1,-1], and we're done.

LP2

- This was used in South Africa to search for new gold mines, where N was the correlation between gold content as function of distance. Look up “Kriging”

- For 1D signals w/ stationary noise, this looks an awful lot like a convolution.

- We get to pick how many points to use in N^{-1} - for large datasets, you probably want to experiment.

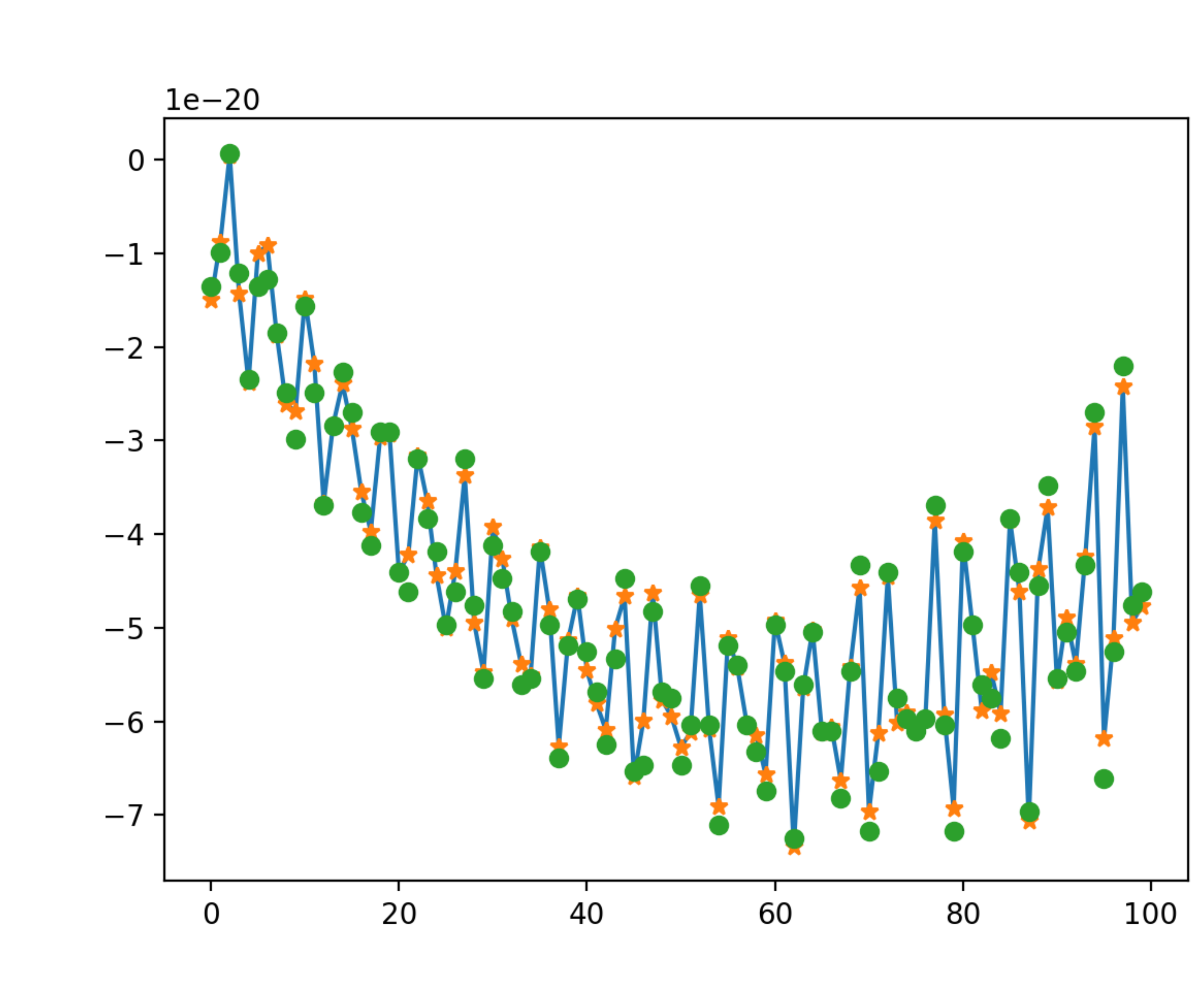
- You may also want to add a (large) offset to N^{-1}. Essentially says you don’t know what the mean is, so don’t push answers towards zero.

LIGO In Practice

- Round raw data (at 64 bits) to some accuracy.
- Find N by taking normalized cross-correlation of LIGO with itself.
- Decide on # of points to use in prediction (~10 seems to work well). Find LP coefficients by inverting 10x10 block of N.
- From 10th point on, predict value given previous 9, save the *difference* between value and prediction.
- To restore, predict value given previous 9, and that to saved value, move one spot to right.
- Goes from 10.3 (raw) to 7.0 (diff) to 3.1 (LP). Bough more than a factor of 2!
- Final expected size is 51 kB, bzip2 gets 63 kB. Compare to 1 MB, we’ve saved factor of 20 with minimal loss.

LP Model

- Blue/yellow stars=true LIGO data
- Green is LP prediction from quantized data.
- We want to save difference between green & yellow.
- Note how green much closer to yellow than yellow neighbors to each other.



Practical Considerations

- Noise is often not quite stationary. In practice, this means you may want to keep # of points small.

- Often, a diff gets within 1/2 byte of optimal. Maybe that's good enough. Not the case for LIGO because of strong lines.

- Good LP has random errors (if they weren't, we could make a better model). Huffman/arithmetic encoding are well matched to this.

- Everything after digitizing/rounding is lossless. Whatever works well is what you should use!

- If you have complicated interpolation/extrapolation, I encourage more investigation into LP. It's often presented opaquely, but if you go back to χ^2, you'll be fine.

- Look at simple_compress_ligo.py for fully worked example.

FLAC

- Flac is another compression utility, designed for lossless compression of music.

- It will work on raw binary data, assuming up to 24-bit samples (annoyingly will not work on 32).

- Uses LP with frequently updated model in case lines change frequency (which is, after all, entire point of music).

- FLAC usually comparable, often better than hand-coded LP. For same quantization, our LP got LIGO to 63655 bytes vs. 66203 for FLAC.

- See compress_ligo_flac.py