cbloom rants

5/01/2018

Secondary Estimation : From PPMZ SEE to PAQ APM

I want to ramble a bit about how Matt Mahoney uses Secondary Estimation in APM's (in his compressors PAQ and BBB), but to get there I thought I'd go through a little story about what secondary estimation is all about.

Secondary Estimation began as a way to apply a (small) correction to a (large) primary model, but we will see in our journey that it has transformed into a way of doing heavy duty modeling itself.

The simplest and perhaps earlier form of secondary estimation may have been bias factors to linear predictors.

Say you are predicting some signal like audio. A basic linear predictor is of the form :

```
to code x[t]

pred = 2*x[t-1] - x[t-2]

delta = x[t] - pred

transmit delta
```

Now it's common to observe that "delta" has remaining predicability. The simplest case is if it just has an overall bias; it tends to be > 0. Then you track the average observed delta and subtract it off to correct the primary predictor. A more interesting case is to use some context to track different corrections. For example :

```
Previous delta as context :

context = delta > 0 ? 1 : 0;

pred = 2*x[t-1] - x[t-2]
delta = x[t] - pred

bias = secondary_estimation[ context ]

transmit (delta - bias)

secondary_estimation[ context ].update(bias)
```

Now we are tracking the average bias that occurs when previous delta was positive or negative, so if they tend to occur in clumps we will remove that bias. ("update" means "average in in some way" which we will perhaps revisit).

Another possibility is to use local shape information, something like :

```
for last 4 values , track if delta was > 0
each of those is a bit
this makes a 4 bit context

look up bias using this context
```

Now you can detect patterns like the transition from flats to edges and apply different biases in each case.

The fundamental thing that we are trying to do here, which is a common theme is :

The primary model is a mathematical predictor, which extracts some fundamental information about the data.

The secondary estimator tracks how well that primary model is fitting the current observation set and corrects for it.

In CALIC a form of secondary estimation very similar to this is used. CALIC uses a shape-adaptive gradient predictor (DPCM lossless image coding). This is the primary model; essentially it tries to fix the local pixels to one of a few classes (smooth, 45 degree edge, 90 degree edge, etc.) and uses different linear predictors for each case. The prediction error from that primary model is then corrected using a table lookup in a shape context. The table lookup tracks the average bias in that shape context. While the primary model uses gradient predictors that are hard-coded (don't vary from image to image), the secondary model can correct for how well those gradient predictors fit the current image.

I'm going to do another hand wavey example before we get into modern data compression.

A common form of difficult prediction that everyone is familiar with is weather prediction.

Modern weather prediction is done using ensembles of atmospheric models. They take observations of various atmospheric variables (temperature, pressure, vorticity, clouds, etc.) and do a physical simulation to evolve the fluid dynamics equations forward in time. They form a range of possible outcomes by testing how variations in the input lead to variations in the output; they also create confidence intervals by using ensembles of slightly different models. The result is a prediction with a probability for each possible outcome. The layman is only shown a simplified prediction (eg. predict 80 degrees Tuesday) but the models actually assign a probability to each possibility (80 degrees at 30% , 81 degrees at 15%, etc.)

Now this physically based simulation is a primary model. We can adjust it using a secondary model.

Again the simplest form would be if the physical simulation just has a general bias. eg. it predicts 0.5 degrees too warm on average. A more subtle case would be if there is a bias per location.

```
primary = prediction from primary model

context = location

track (observed - primary)

store bias per context

secondary = primary corrected by bias[context]
```

Another correction we might want is the success of yesterday's prediction. You can imagine that the prediction accuracy is usually better if yesterday's prediction was spot on. If we got yesterday grossly wrong, it's more likely we will do so again. Now in the primary physical model, we might be feeding in yesterday's atmospheric conditions as context already, but it is not using yesterday's prediction. We are doing secondary estimation using side data which is not in the primary model.

You might also imagine that there is no overall bias to the primary prediction, but there is bias depending on what value it output. And furthermore that bias might depend on location. And it might depend on how good yesterday's prediction was. Now our secondary estimator is :

```
make primary prediction

context = value of primary, quantized into N bits

(eg. maybe we map temperature to 5 bits, chance of precipitation to 3 bits)
```

```
context += location
context += success of yesterday's prediction

secondary = primary corrected by bias[context]
```

Now we can perform quite subtle corrections. Say we have a great overall primary predictor. But in Seattle, when the primary model predicts a temperature of 60 and rain at 10% , it tends to actually rain 20% of the time (and more likely if we got yesterday wrong).

SEE in PPMZ

I introduced Secondary Estimation (for escapes) to context modeling in PPMZ.

(See "Solving the Problems of Context Modeling" for details; I was obviously quite modest about my work back then!)

Let's start with a tiny bit of background on PPM and what was going on in PPM development back then.

PPM takes a context of some previous characters (order-N = N previous characters) and tracks what characters have been seen in that context. eg. after order-4 context "hell" we may have seen space ' ' 3 times and 'o' 2 times (and if you're a bro perhaps 'a' too many times).

Classic PPM starts from deep high order contexts (perhaps order-8) and if the character to be coded has not been seen in that context order, it sends an "escape" and drops down to the next lower context.

The problem of estimating the escape probability was long a confounding. The standard approach was to use the count of characters seen in the context to form the escape probability.

```
A nice summary of classical escape estimators from
"Experiments on the Zero Frequency Problem" :

n = the # of tokens seen so far
u = number of unique tokens seen so far
ti = number of unique tokens seen i times

PPMA : 1 / (n+1)

PPMB : (u - t1) / n

PPMC : u / (n + u)

PPMD : (u/2) / n

PPMP : t1/n - t2/n^2 - t3/n^3 ...

PPMX : t1 / n

PPMXC : t1 / n if t1 < n
        u / (n + u) otherwise
```

eg. for PPMA you just leave the escape count at 1. For PPMC, you increment the escape count each time a novel symbol is seen. For PPMD, you increment the escape count by 1/2 on a novel symbol (and also set the novel characters count to 1/2) so the total always goes up by 1.

Now some of these have theoretical justifications based on Poisson process models and Laplacian priors and so on, but the correspondence of those priors to real world data is generally poor (at low counts).

The big difficult with PPM (and data compression in general) is that low counts is where we do our business. We can easily get contexts with high counts (sufficient statistics) and accurate estimators by dropping to lower orders. eg. if you did PPM and limit your max order to order-2 , you wouldn't have this problem. But we generally want to push our orders to the absolute highest breaking point, where our statistical density goes to shit. Big compression gains can be found there, so we need to be able to work where we may have only seen 1 or 2 events in a context before.

Around the time I was doing PPMZ, the PPM* and PPMD papers by Teahan came out ("Unbounded length contexts for PPM" and "Probability estimation for PPM"). PPMD was a new better estimator, but to me it just screamed that something was wrong. All these semi-justified mathematical forms for the escape probability seemed to just be the wrong approach. At the same time, PPM* showed the value of long deterministic contexts, where by definition "u" (the number of unique symbols) is always 1 and "n" (total) count is typically low.

We need to be able to answer a very important and difficult question. In a context that has only seen one symbol one time, what should the escape probability be? (equivalently, what is the probability that one symbol is right?). There is no simple formula of observed counts in that context that can solve that.

Secondary Estimation is the obvious answer. We want to know :

```
P(esc) when only 1 symbol has been seen in a context
```

We might first observe that this P(esc) varies between files. On more compressible files, it is much lower. On near-random files, it can be quite high. This is a question of to what extent is this novel deterministic context a reflection of the end statistics, or just a mirage due to sparse statistics.

That is,

```
In an order-8 context like "abcdefgh" we have only even seen the symbol 'x' once

Is that a reflection of a true pattern?  That "abcdefgh" will always predict 'x' ?

Or is the following character in fact completely random, and if we saw more symbols we
would see all of them with equal probability
```

Again you don't have the information within the context to tell the difference. Using an average across the whole file is a very rough guess, but obviously you could do better.

You need to use out-of-context observations to augment the primary model. Rather than just tracking the real P(esc) for the whole file we can obviously use some context to find portions of the data where it behaves differently. For example P(esc) (when only 1 symbol has been seen in a context) might be 75% if the parent context has high entropy; it might be 50% if the order is 6, it might be 10% if order is 8 and the previous symbols were also coded from high order with no escape.

In PPMZ, I specifically handled only low escape counts and totals; that is, it was specifically trying to solve this sparse statistics problem. The more modern approach (APM's, see later) is to just run all probabilities through secondary statistics.

The general approach to secondary estimation in data compression is :

```
create a probability from the primary model

take {probability + some other context} and look that up in a secondary model

use the observed statistics in the secondary model for coding
```

The "secondary model" here is typically just a table. We are using the observed statistics in one model as the input to another model. In PPMZ, the primary model is large and sparse while the secondary model is small and dense, but that need not always be the case.

The secondary model is usually initialized so that the input probabilities pass through if nothing has been observed yet. That is,

```
initially :

secondary_model[ probability + context ] = probability
```

Secondary estimation in cases like PPM can be thought of as a way of sharing information across sparse contexts that are not connected in the normal model.

That is, in PPM contexts that are suffixes have a parent-child relationship and can easily share information; eg. "abcd" and "bcd" and "cd" are connected and can share observations. But some other context "xyzw" is totally unconnected in the tree. Despite that, by having the same statistics they may be quite related! That is

```
"abcd" has seen 3 occurances of symbol 'd' (and nothing else)

"xyzw" has seen 3 occurances of symbol 'd' (and nothing else)
```

these are probably very similar contexts even though they have no connection in the PPM tree. The next coding event that happens in either context, we want to communicate to the other. eg. say a symbol 'e' now occurs in "abcd" - that makes it more likely that "xyzw" will have an escape. That is, 3-count 'd' contexts are now less likely to be deterministic. The secondary estimation table allows us to accumulate these disparate contexts together and merge their observations.

The PPMZ SEE context is made from the escape & total count, as well as three different orders of previous symbol bits. The three orders and then blended together (an early form of weighted mixing!). I certainly don't recommend the exact details of how PPMZ does it today. The full details of the PPMZ SEE mechanism can be found in PPMZ (Solving the Problems of Context Modeling) (PDF) . You may also wish to consult the later work by Shkarin "PPM: one step to practicality" (PPMd/PPmonstr/PPMii) which is rather more refined. Shkarin adds some good ideas to the context used for SEE, such as using the recent success so that switches between highly compressible and less compressible data can be modeled.

Let me repeat myself to be clear. Part of the purpose & function of secondary estimation is to find patterns where the observed counts in a state to do not linearly correspond to the actual probability in any way.

That is, secondary estimation can model things like :

```
Assuming a binary coder
The coder sees 0's and 1's
Each context state tracks n0 and n1 , the # or 0's and 1's seen in that context

On file A :

if observed { n0 = 0, n1 = 1 } -> actual P1 is 60%
if observed { n0 = 0, n1 = 2 } -> actual P1 is 70%
if observed { n0 = 0, n1 = 3 } -> actual P1 is 90%

On file B :

if observed { n0 = 0, n1 = 1 } -> actual P1 is 70%
if observed { n0 = 0, n1 = 2 } -> actual P1 is 90%
if observed { n0 = 0, n1 = 3 } -> actual P1 is 99%
```

```
The traditional models are of the form :

P1 = (n1 + c) / (n0 + n1 + 2c)

and you can make some Bayesian prior arguments to come up with different values of c (1/2 and 1 are common)
```

The point is there is *no* prior that gets it right. If the data actually came from a Laplacian or Poisson source, then observing counts you could make estimates of the true P in this way. But context observations do not work that way.

There are lots of different effects happening. One is that there is multiple "sources" in a file. There might be one source that's pure random, one source is purely predictable (all 0's or all 1's), another source is in fact pretty close to a Poisson process. When you have only a few observations in a context, part of the uncertainty is trying to guess which of the many sources in the file that context maps to.

Adaptive Probability Map (APM) ; secondary estimation & beyond.

Matt Mahoney's PAQ (and many other modern compressors) make heavy use of secondary estimation, not just for escapes, but for every coding event. Matt calls it an "APM" , and while it is essentially the same thing as a secondary estimation table, the typical usage and some details are a bit different, so I will call them APMs here to distinguish.

PPM-type compressors hit a dead end in their compression ratio due to the difficult of doing things like mixing and secondary estimation in character alphabets. PAQ, by using exclusively binary coding, only has one probability to work with (the probability of a 0 or 1, and then the other is inferred), so it can be easily transformed. This allows you to apply secondary estimation not just to the binary escape event, but to all symbol coding.

An APM is a secondary estimation table. You take a probability from a previous stage, look it up in a table (with some additional context, optionally), and then use the observed statistics in the table, either for coding or as input to another stage.

By convention, there are some differences in typical implementation choices. I'll describe a typical APM implementation :

```
probability P from previous stage

P is transformed nonlinearly with a "stretch"

[stretch(P) + context] is looked up in APM table

observed P there is passed to next stage

optionally :

look up both floor(P) and ceil(P) in some fixed point to get two adjacent APM entries
linearly interpolate them using fractional bits
```

A lot of the details here come from the fact that we are passing general P's through the APM, rather than restricting to only low counts as in PPMZ SEE. That is, we need to handle a wide range of P's.

Thus, you might want to use a fixed point to index the APM table and then linearly interpolate (standard technique for tabulated function lookup); this lets you use smaller, denser tables and still get fine precision.

The "stretch" function is to map P so that we quantize into buckets where we need them. That is, if you think in terms of P in [0,1] we want to have smaller buckets at the endpoints. The reason is that P steps near 0 or 1 make a very large difference in codelen, so getting them right there is crucial. That is, a P difference of 0.90 to 0.95 is much more important than from 0.50 to 0.55 ; instead of having variable bucket sizes (smaller buckets at the end) we use uniform quantization but stretch P first (the ends spread out more). One nice option for stretch is the "logit" function.

```
LPS symbol codelen (LPS = less probable symbol)

steps of P near 0.5 produce small changes in codelen :

P = 0.5  : 1 bit
P = 0.55 : 1.152 bits

steps of P near 1.0 produce big changes in codelen :

P = 0.9  : 3.32193 bits
P = 0.95 : 4.32193 bits
```

The reason why you might want to do this stretch(P) indexing to the APM table (rather than just look up P) is again density. With the stretch(P) distortion, you can get away with only 5 or 6 bits of index, and still get good resolution where you need it. With linear P indexing you might need a 10 bit table size for the same quality. That's 32-64 entries instead of 1024 which is a massive increase in how often each slot gets updated (or a great decrease in the average age of statistics; we want them to be as fresh as possible). With such course indexing of the APM, the two adjacent table slots should be used (above and below the input P) and linearly interpolated.

APM implementations typically update the observed statistics using the "probability shift" method (which is equivalent to constant total count or geometric decay IIR).

An APM should be initialized such that it passes through the input probability. If you get a probability from the previous stage, and nothing has yet been observed in the APM's context, it passes through that probability. Once it does observe something it shifts the output probability towards the observed statistics in that context.

We can see something interesting that APM's can do already :

I wrote before about how secondary estimation is crucial in PPM to handle sparse contexts with very few events. It allows you to track the actual observed rates in that class of contexts. But we thought of SEE as less important in dense contexts.

That is true only in non-time-varying data. In the real world almost all data is time varying. It can shift character, or go through different modes or phases. In that sense all contexts are sparse, even if they have a lot of observed counts, because they are sparse in observation of recent events. That is, some order-4 context in PPM might have 100 observed counts, but most of those were long in the past (several megabytes ago). Only a few are from the last few bytes, where we may have shifted to a different character of data.

Running all your counts through an APM picks this up very nicely.

```
During stable phase :

look up primary model

secondary = APM[ primary ]

when primary is dense, secondary == primary will pass through


Now imagine we suddenly transition to a chunk of data that is nearly random

the primary model still has all the old counts and will only slowly learn about the new random data

but the APM sees every symbol coded, so it can learn quickly
```

```
APM[] will quickly tend towards APM[P] = 0.5 for all P

that is, the input P will be ignored and all states will have 50/50 probability
```

This is quite an extreme example, but more generally the APM can pick up regions where we move to more or less compressible data. As in the PPM SEE case, what's happening here is sharing of information across parts of the large/sparse primary model that might not otherwise communicate.

Let me repeat that to be clear :

Say you have a large primary model, with N context nodes. Each node is only updated at a rate of (1/N) times per byte processed. The APM on the other hand is updated every time. It can adapt much faster - it's sharing information across all the nodes of the primary model. You may have very mature nodes in the primary model with counts like {n0=5,n1=10} (which we would normally consider to be quite stable). Now you move to a region where the probability of a 1 is 100%. It will take *tons* of steps for the primary model to pick that up and model it, because the updates are scattered all over the big model space. Any one node only gets 1/N of the updates, so it takes ~N*10 bytes to get good statistics for the change.

An interesting common way to use APM's is in a cascade, rather than a single step with a large context.

That is, you want to take some P from the primary model, and you have additional contexts C1 and C2 to condition the APM step. You have two options :

```
Single step, large context :

P_out = APM[ {C1,C2,P} ]

Two steps, small context, cascade :

P1 = APM1[ {C1,P} ]

P_out = APM2[ {C2,P1} ]
```

That is, feed P through an APM with one context, then take that P output and pass it through another APM with a different context, as many times as you like.

Why use a cascade instead of a large context? The main reason is density.

If you have N bits of extra context information to use in your APM lookup, the traditional big table approach dilutes your statistics by 2^N. The cascade approach only dilutes by *N.

The smaller tables of the APM cascade mean that it cannot model certain kinds of correlations. It can only pick up ways that each context correlated to biases on P. It cannot pick up joint context correlations.

```
The APM cascade can model things like :

if C1 = 0 , high P's tend to skew higher  (input P of 0.7 outputs 0.9)

if C2 = 1 , low P's tend to skew lower

But it can't model joint things like :

if C1 = 0 and C2 = 1 , low P's tend towards 0.5
```

One nice thing about APM cascades is that they are nearly a NOP when you add a context that doesn't help. (as opposed to the single table large context method, which has a negative diluting effecting when you add context bits that don't help). For example if C2 is just not correlated to P, then APM2 will just pass P through unmodified. APM1 can model the correlation of C1 and P without being molested by the mistake of adding C2. They sort of work independently and stages that aren't helping turn themselves off.

One funny thing you can do with an APM cascade is to just use it for modeling with no primary model at all. This looks like :

```
Traditional usage :

do order-3 context modeling to generate P
modify observed P with a single APM :
P = APM[P]

APM Cascade as the model :

P0 = order-0 frequency
P1 = APM1[ o1 | P0 ];
P2 = APM2[ o2 | P1 ];
P3 = APM3[ o3 | P2 ];
```

That is, just run the probability from each stage through the next to get order-N modeling.

We do this from low order to high, so that each stage can add its observation to the extent it has seen events. Imagine that the current order2 and order3 contexts have not been observed at all yet. Then we get P1 from APM1, pass it through APM2, since that has seen nothing it just passes P1 through, then APM3 does to. So we wind up getting the lowest order that has actually observed things.

This method of using an APM cascade is the primary model is used by Mahoney in BBB.

A 2d APM can also be used as a mixer, which I think I will leave for the next post.

cbloom at 11:58 AM