Some Observations

Daniel Seita

October 23, 2015

1 Convegence of BayesNet on Student Data

1.1 Norm Difference for 50% Data

Here are the settings I used for this subsection, with the SAME variable set to one and obviously changing each time. I ran these on `bitter`. Note: I used 25001 here, but later, I switched to more "even" batch sizes.

```
Option Name       Type          Value
===========       ====          =====
addConstFeat      boolean       false
autoReset         boolean       true
batchSize         int           25001
copiesForSAME     int           1
cumScore          int           0
debug             int           0
debugMem          boolean       false
dim               int           5
doubleScore       boolean       false
evalStep          int           11
featThreshold     Mat           null
featType          int           1
isprob            boolean       false
npasses           int           200
numSamplesBurn    int           0
nzPerColumn       int           0
power             float         0.0
pstep             float         0.01
putBack           int           1
resFile           String        null
sample            float         1.0
samplingRate      int           1
sizeMargin        float         3.0
startBlock        int           8000
updateAll         boolean       false
useCache          boolean       true
useDouble         boolean       false
useGPU            boolean       true
warmup            long          0
```

Figures 1 and 2 plot the norm differences after every fourth batch for the first 500 of the 7000 total data points. (There are too many data points to visualize easily.) Here was the code to generate those figures:

```
>>> plt.plot(n01r[idx],'ro-',n02r[idx],'y^-',n03r[idx],'bs-',n04r[idx],'cv-',n05r[idx],'k*-')
>>> plt.ylim([0,0.5])
>>> plt.xlim([0,125])
```

So red corresponded to $m = 1$, then yellow, blue, cyan, and black for the others. We can see that the SAME advantage is clear at the very beginning, and as we increase m, it looks like we get diminishing returns.

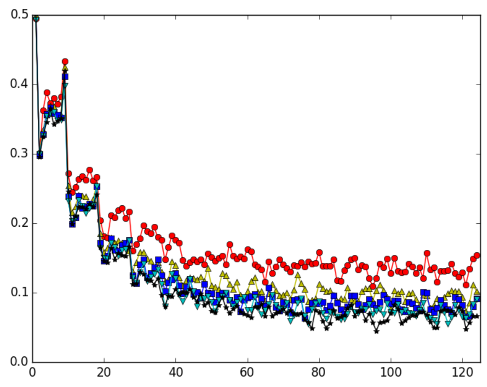

Figure 1: Same seeds and norm differences for $m = \{1, 2, 3, 4, 5\}$.

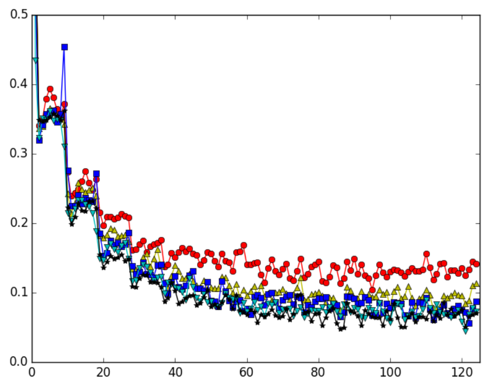

Figure 2: Different seeds and norm differences for $m = \{1, 2, 3, 4, 5\}$.

Here is the *long-run*. The previous two figures only sub-sampled data from the first two npasses or so (out of 200 total). Figures 3 and 4 show the data in the long run over 200 npasses. I subsampled once every 35 points, which turned out to give me one data point for each npass. These figures show that in the long run, the norm differences seem to converge to different quantities.

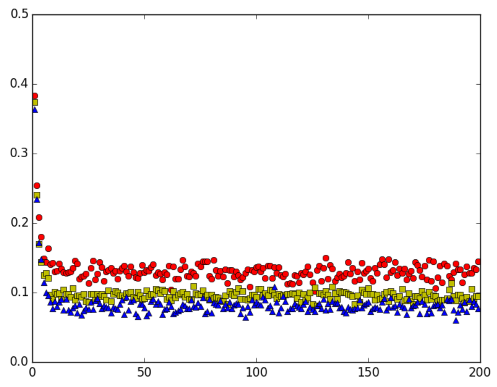

Figure 3: Same seeds and norm differences for $m = \{1, 2, 3\}$.

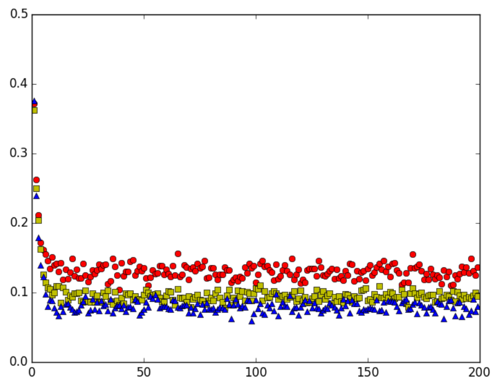

Figure 4: Different seeds and norm differences for $m = \{1, 2, 3\}$.

Note that for all the figures here, I typically truncate the y-axis to make some of the differences more visible. Here was the code for the last two where I sampled over all 7000 dadta points from the 200 npasses:

```
>>> idx = [i for i in range(0,7000,35)]
>>> plt.plot(n01r[idx],'ro',n02r[idx],'ys',n03r[idx],'b^')
>>> plt.ylim([0,0.5])
```

1.2 KL-Divergence for 50% Data

Figures 5 and 6 show the average KL divergences, and show that there is a similar trend. This was with equal subsampling over the entire interval of 200 npasses.

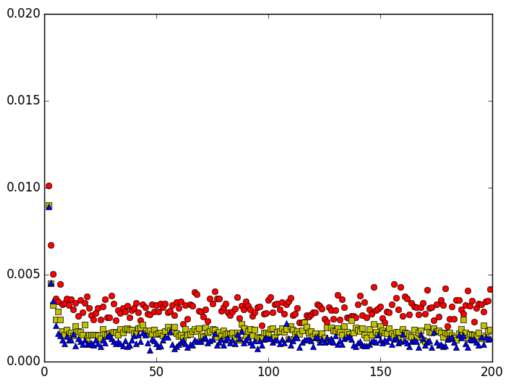

Figure 5: Same seeds and average KL divergences for $m = \{1, 2, 3\}$.

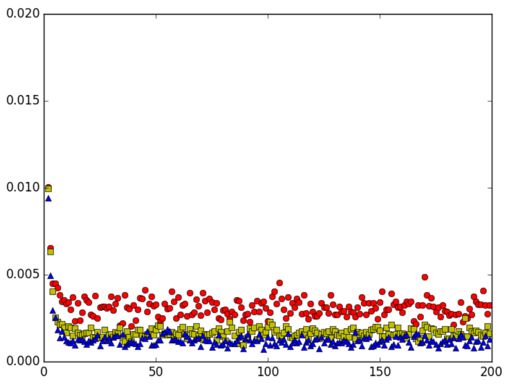

Figure 6: Different seeds and average KL divergences for $m = \{1, 2, 3\}$.

1.3 Running Time and Gflops

Here is the plot of times and gflops, in case you are interested, but these should be taken with a grain of salt.

```
Some timing results from BIDMach.

(A) Here are the times and gflops with 200 passes, batch size 25001, SAME from 1 to
11, same random seed. I was printing out the norm differences, but NOT the KL divergences,
which explains the difference between the times here and in section (B), which are
otherwise the same (well, I didn't do 11 there...).  I am using the 50% data.

Time=082.6820 secs, gflops=1.69 (1)
Time=107.8820 secs, gflops=2.59 (2)
Time=117.8430 secs, gflops=3.55 (3)
Time=126.6750 secs, gflops=4.41 (4)
Time=136.3580 secs, gflops=5.12 (5)
Time=144.9030 secs, gflops=5.78 (6)
Time=156.2770 secs, gflops=6.25 (7)
Time=164.0710 secs, gflops=6.80 (8)
Time=174.6650 secs, gflops=7.19 (9)
Time=183.6360 secs, gflops=7.60 (10)
Time=192.2530 secs, gflops=7.98 (11)

(B) Here are the times and gflops with 200 passes, batch size 25001, SAME from 1 to 10,
different random seeds, and printing out KL and norms (so that takes time). I ran this on
bitter, so hopefully John wasn't using it.  I am using the 50% data.

Time=105.6010 secs, gflops=1.32 (1)
Time=130.4010 secs, gflops=2.14 (2)
Time=140.8700 secs, gflops=2.97 (3)
Time=151.6070 secs, gflops=3.68 (4)
Time=160.5840 secs, gflops=4.35 (5)
Time=167.9420 secs, gflops=4.99 (6)
```

```
Time=179.8230 secs, gflops=5.43 (7)
Time=187.3620 secs, gflops=5.96 (8)
Time=197.6430 secs, gflops=6.35 (9)
Time=206.8040 secs, gflops=6.75 (10)

(C) Same as (B), but with a different random seed each time.

Time=105.5740 secs, gflops=1.32 (1)
Time=131.6690 secs, gflops=2.12 (2)
Time=140.6400 secs, gflops=2.98 (3)
Time=150.1360 secs, gflops=3.72 (4)
Time=159.1510 secs, gflops=4.38 (5)
Time=168.0350 secs, gflops=4.98 (6)
Time=179.0620 secs, gflops=5.46 (7)
Time=186.9820 secs, gflops=5.97 (8)
Time=198.5040 secs, gflops=6.33 (9)
Time=205.8420 secs, gflops=6.78 (10)
```

To summarize: the times here appear to be good, suggesting that adding in more m samples results in a simple linear increase in runtime, and for small m that is dominated by the other stuff we do (e.g., printing out to file, initialization, etc.).

1.4 Mirroring the SAME Paper

Now I will attempt to follow the SAME paper's plots. Figures 7 and 8 show my efforts. These show the log likelihood based on the number of passes for the same student data, with 5 different SAME parameters $m \in \{1, 5, 10, 20, 50\}$. All of these used the same batch size of 50,000 columns (so the one with $m = 50$ was running low on GPU memory, but we can use a smaller batch for higher m). It says 60 on the axis, but that is because there were three log likelihood tests per iteration.

The first of these uses the same random seed for all five; the second uses different random seeds in order to diversify the results. Unfortunately, SAME does not appear to significantly improve log likelihood convergence, though perhaps this data is too simple to analyze that. Note: apologies for the lack of a legend here, but a legend would not have been helpful anyway, given how bunched-up things appear. The code for that in `matplotlib` is

```
plt.plot(d01r,'y^',d05r,'b-',d10r,'r+',d20r,'k--',d50r,'g*')
```

So that kind of tells us the legend.

I also investigated runtime for the second case with different seeds (the other one was similar).

```
Time=6.0520 secs, gflops=2.32 (for m = 1)
Time=9.6240 secs, gflops=7.31 (for m = 5)
Time=14.3540 secs, gflops=9.80 (for m = 10)
Time=23.4500 secs, gflops=11.99 (for m = 20)
Time=51.2570 secs, gflops=13.72 (for m = 50)
```

We can see that it does takes longer per iteration. This seems to contradict that SAME paper, unfortunately? **UPDATE: No, it doesn't my apologies.** Again, these were straight from the experiments in this section, which used the same batch size.

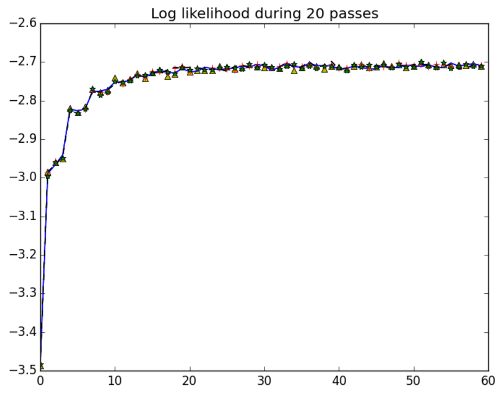

Figure 7: Same seeds and l.l.

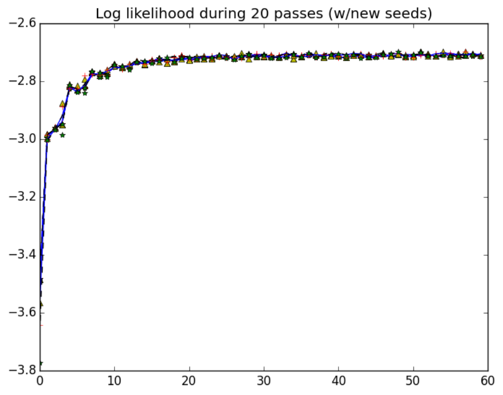

Figure 8: Different seeds and l.l.

2 Updated Results (For 10/23 Meeting)

Here are some extra results for the 10/23 meeting, based on some additional feedback. First, I am only going to be using the "average KL divergence" metric from now on (at least for the five variable student data). Second, I will be using a different random seed for each value of the SAME parameter, to introduce more diversity at the beginning (but it shouldn't noticeably affect the results). Third, I will be testing with different sparsity levels.

Here were the settings for the experiments in this section:

```
Option Name       Type          Value
===========       ====          =====
addConstFeat      boolean       false
autoReset         boolean       true
batchSize         int           24000
copiesForSAME     int           1
cumScore          int           0
debug             int           0
debugMem          boolean       false
dim               int           5
doubleScore       boolean       false
evalStep          int           11
featThreshold     Mat           null
featType          int           1
isprob            boolean       false
npasses           int           10
numSamplesBurn    int           0
nzPerColumn       int           0
power             float         0.0
pstep             float         0.01
putBack           int           1
resFile           String        null
sample            float         1.0
samplingRate      int           1
sizeMargin        float         3.0
startBlock        int           8000
updateAll         boolean       false
useCache          boolean       true
useDouble         boolean       false
useGPU            boolean       true
warmup            long          0
```

First, let's look at the average KL divergence for the five-variable student data with varying levels of m (not consecutive!) and varying sparsity levels. Figures 9, 10, and 11 show the KL divergence between what BayesNet gets and what the true distribution is, for 50 percent, 25 percent, and 10 percent of the data *known*, respectively (i.e., this is in order of increasing sparsity).

The m used were $m \in \{1, 5, 10, 20, 50\}$. Note that these all had a batch size of 24000. Using $m = 100$ resulted in an out of memory error, so one would need to decrease the batch size for that. For all three of the subsequent plots, I used data from ten passes[1], subsampling equally within that range to make the graphs readable. (That is how to interpret the x-axis of the plots.) I subsampled every fourth element.

The code to generate these figures was:

```
>>> d01 = np.loadtxt('same_01_50.txt')
```

[1]The algorithm tends to converge after only three iterations, and as discussed in earlier sections, higher m values result in better long-term performance.

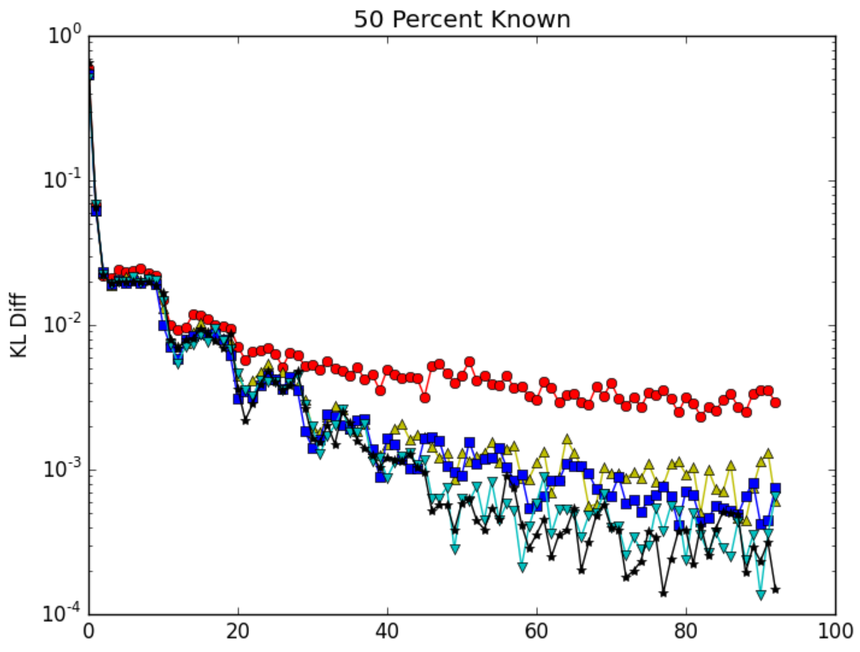

Figure 9: Average KL divergence difference of the five variable student data with 50 percent of the data known. We are using a log scale for readability purposes.

```
...(etc)...
>>> d50 = np.loadtxt('same_50_50.txt')
>>> plt.plot(d01[::2],'ro-',d05[::2],'y^-',d10[::2],'bs-',d20[::2],'cv-',d50[::2],'k*-')
>>> plt.yscale('log')
>>> plt.title('50 Percent Known')
>>> plt.ylabel('KL Difference')
>>> plt.savefig('kl_student_50perc.png')
```

Therefore, the legend is that $m = \{1, 5, 10, 20, 50\}$ is, in order, red, yellow, blue, cyan, and black. One can clearly see for all three sparsity levels that the advantage of the initial first few m, but then there are diminishing returns.

The 50 and 25 percent known data behave "best" since their plots seem to converge nicely. Figure 11 is a little more weird, and my conclusion is that with very sparse data, it is harder for the algorithm to converge, and we would need to have a stronger prior perhaps to regulate this behavior. For all three earlier figures in this section, I am only using a uniform Dirichlet prior, with scale 1, and a value of 1 for each component (so the "count" is not very strong).

Despite the weird results for the 10 percent known case, realize that the resulting CPT *is not actually that bad*. After 10 iterations with the 10 percent known data, with $m = 1$ (i.e., the worst case!), we get the following CPT:

```
CPT for node indexed at 0
Parents:
        0.7187 0.2813

CPT for node indexed at 1
Parents:
        0.5568 0.4432
```

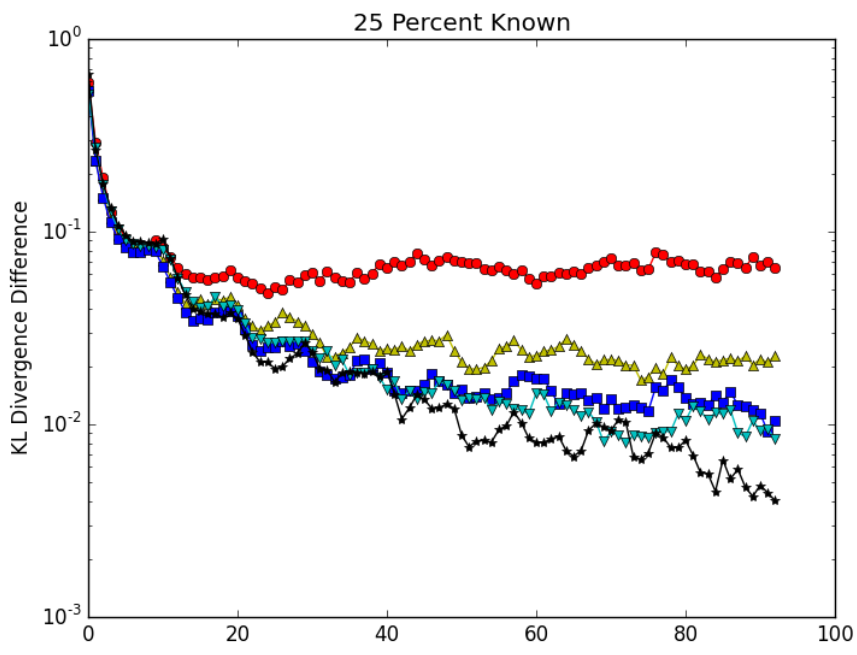

Figure 10: Average KL divergence difference of the five variable student data with 25 percent of the data known. We are using a log scale for readability purposes.

```
CPT for node indexed at 2
Parents: 0
0       0.8378 0.1622
1       0.4486 0.5514

CPT for node indexed at 3
Parents: 0,1
0 0     0.3318 0.4447 0.2235
0 1     0.0000 0.3287 0.6712
1 0     0.5864 0.4135 0.0001
1 1     0.5348 0.0026 0.4626

CPT for node indexed at 4
Parents: 3
0       0.3166 0.6834
1       0.0119 0.9881
2       0.9999 0.0001
```

This is not terrible and given how sparse the data is, it might be fine. Node 0, for instance should be 70-30, and we get something similar. Node 1 should be 60-40, and we get something that *resembles* that we are getting there. In general, we do see most CPTs having roughly the same values, with only a handful of major differences.

Next, for the 25 percent known data, along with the same settings as before, I got the following time results. These were done during the same time frame, so I do not think there was anything on the `bitter` machine that would have affected these results.

```
Time=4.8230 secs, gflops=1.46   (m = 1)
Time=7.4520 secs, gflops=4.73   (m = 5)
Time=9.8670 secs, gflops=7.15   (m = 10)
```

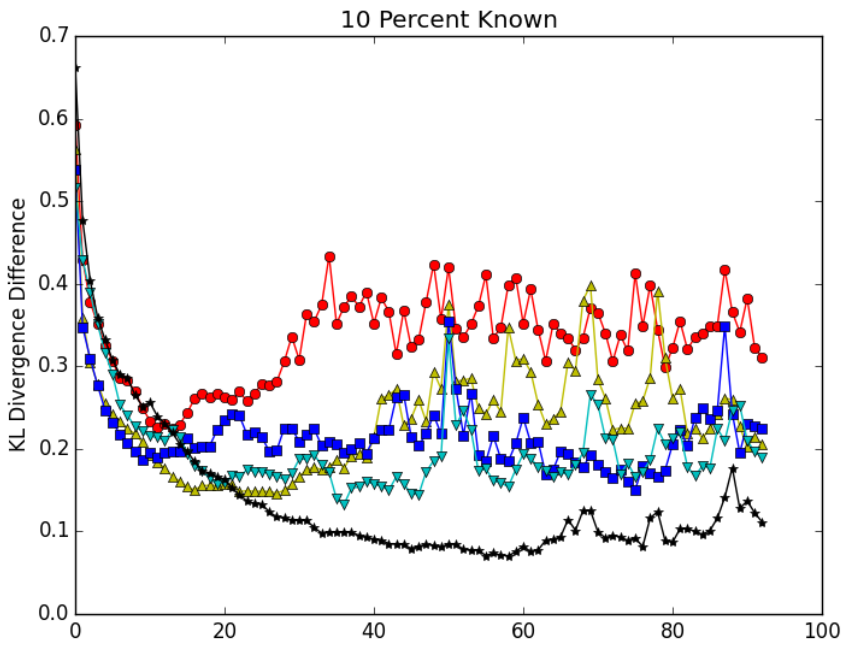

Figure 11: Average KL divergence difference of the five variable student data with 10 percent of the data known. This is *not* on a log scale.

```
Time=14.4160 secs, gflops=9.79  (m = 20)
Time=18.8420 secs, gflops=11.23 (m = 30)
Time=23.7510 secs, gflops=11.88 (m = 40)
Time=28.4250 secs, gflops=12.41 (m = 50)
Time=32.5830 secs, gflops=12.99 (m = 60)
```

There does seem to be some saturation as we get up to the high m values here. For instance, the gflops goes from 11.23 to 12.99, but the corresponding runtime is 18.84 seconds to 32.58 seconds – a much larger relative change. Again, I used the same batch size for these (24000) and I would have tested with larger m values, except that $m = 60$ was using all but eight percent of the memory, and lowering the batch size lowers the gflops. As an example, if I use $m = 70$ with a batch size of 16000, the time is 39.774 seconds (slower than any of the above results, which makes sense) and the gflops is 12.41.

Next, I investigated how much the raw times matter. Using higher m values is fine, *except* if it causes too much time delay. Therefore, what I did was run experiments with five different m values, but at the *end* of each pass over the data, I would record the time, along with the corresponding average KL divergence difference for that particular batch. In other words, for each batch, the last one is used to measure statistics such as log likelihood. I took the time reported there, and then computed the average KL divergence difference. Figures 12 and 13 plot this information, with the former using the raw values, and the latter using a log scale on the y axis.

Note that for each of these plots, one "symbol" indicates one pass over the data, so the $m = 50$ curve only has ten "stars" because there were only ten passes, whereas the others have denser curves. I cut off the curves when appropriate to make the plots readable.

What can we conclude? With $m = 1$, its performance is very bad and having a fast runtime only matters in the very beginning. For instance, after five seconds, the $m = 50$ case already

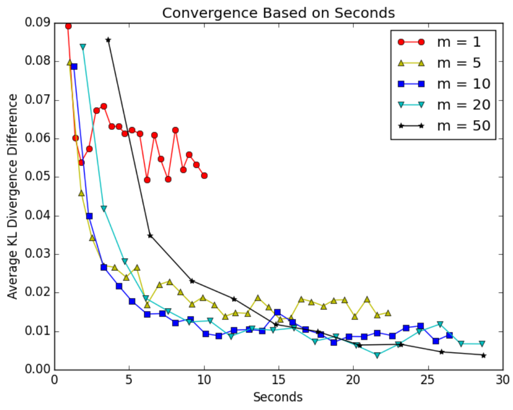

Figure 12: Raw KL Divergences

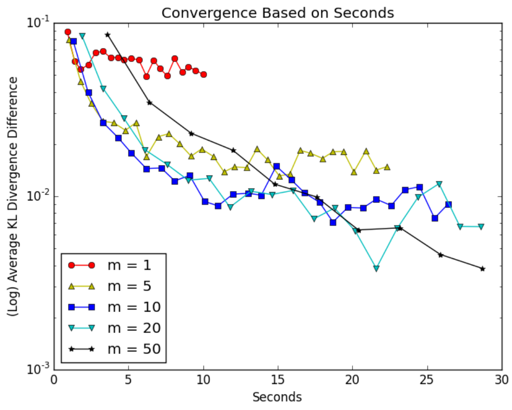

Figure 13: Log-scaled KL Divergences

beats it (according to our average KL divergence metric). We also see that, at around 15 seconds, the best appears to be around $m = 10$ or $m - 20$, but beyond that point, $m = 50$ reigns superior on a per-time basis.

At least for this data, higher m is definitely worth it, but perhaps if there is other data that has a longer startup time, things might favor smaller m. Again, I used the 25 percent data, with a fixed batch size of 24000 for all of these (except 16000 for the last one, which was used to compute these statistics here).

To generate the code, I had to use the following set of commands:

```
>>> d01 = np.loadtxt('same_time_01.txt', unpack=True)
...(etc)...
>>> d50 = np.loadtxt('same_time_50.txt', unpack=True)
>>> plt.plot(d01[1],d01[0],'ro-',label="m = 1")
>>> plt.plot(d05[1],d05[0],'y^-',label="m = 5")
>>> plt.plot(d10[1],d10[0],'bs-',label="m = 10")
>>> plt.plot(d20[1],d20[0],'cv-',label="m = 20")
>>> plt.plot(d50[1],d50[0],'k*-',label="m = 50")
>>> plt.legend(loc='upper right')
>>> plt.savefig('kl_time_raw.png')
>>> plt.ylabel('Average KL Divergence Difference')
>>> plt.xlabel('Seconds')
>>> plt.title('Convergence Based on Seconds')
>>> plt.savefig('kl_time_raw.png')
```

Note the use of the unpack command, since this time we want to *jointly* plot (x, y) data. Also, I *finally* figured out how to get legends up there nicely ... I didn't realize that they were that easy.