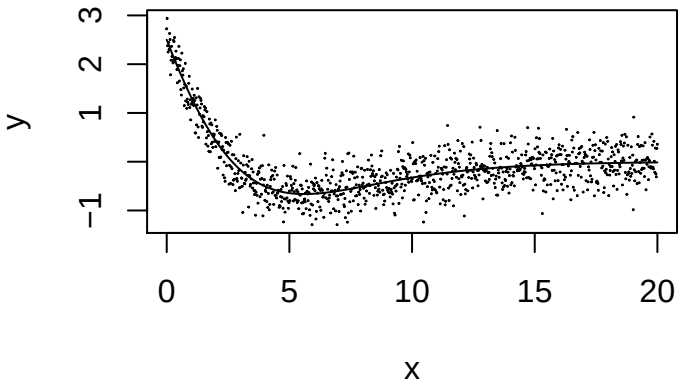
STAT 9100 Homework 1

Peng Shao

January 27, 2016

Problem 1

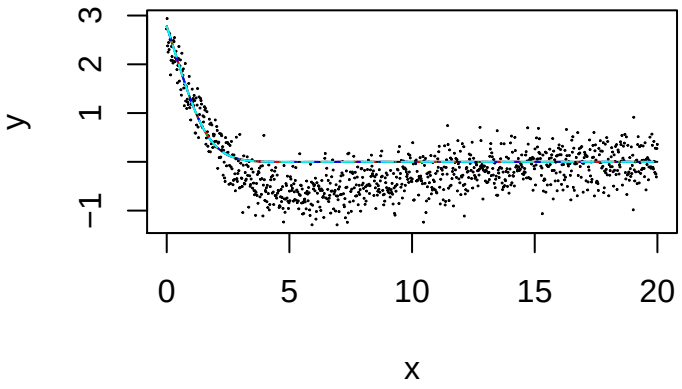
a).



b).

For different starting values, by default, I set $\alpha = 0.5$ and $m = \frac{\text{number of observations}}{5}$. The results are list as below

	theta0	theta1	theta2	theta3	SSE	n.iter
(4.1, -1.4, 0.65, 0.35)	3.648	-0.852	0.312	1.388	205.451	604
(20, 20, 20, 20)	3.464	-0.499	0.276	1.628	206.170	2037
(-20, -20, -20, -20)	10.172	0.642	2.587	1.246	211.174	2910
(0, 0, 0, 0)	3.978	-0.766	0.365	1.554	207.696	124
(4, -1.5, 0.6, 0.4)	9.597	0.069	4.337	0.893	239.218	1332



From the table, it seems that β's are totally different based on different starting values. We may think that the results are sensitive to the starting values. However, from the graph we can see that all results produce an identical curve, which means that the procedure is not so sensitive to the starting values. Since the starting

values are not the problem, then the reason which causes the algorithm converge to a local minima instead of true curve should be either the learning rate or the batch size or both.

From the second table and the third table, we can see that when α is less than 0.3 and m is greater than 100, the result can be very stable and precise, even the number of iteration is not that stable. Thus the algorithm seems useful.

	theta0	theta1	theta2	theta3	SSE	n.iter
alpha=0.1	4.096	-1.556	0.616	0.403	96.144	874
alpha=0.2	4.223	-1.619	0.655	0.409	96.253	3547
alpha=0.3	4.061	-1.581	0.521	0.440	97.542	3917
alpha=0.4	3.872	-1.068	0.348	1.311	205.032	3949
alpha=0.5	3.894	-1.081	0.406	1.206	203.878	396
alpha=0.6	2.960	0.145	0.202	1.905	205.649	4277
alpha=0.7	3.035	0.393	0.240	1.919	205.778	3540
alpha=0.8	4.270	-0.026	2.900	16.801	415.194	32
alpha=0.9	3.169	0.229	0.267	1.818	205.879	2566

	theta0	theta1	theta2	theta3	SSE	n.iter
m=1	4.080	-1.415	0.728	0.996	209.587	19
m=10	4.146	-1.353	0.669	1.489	226.003	2290
m=100	4.105	-1.564	0.641	0.400	96.144	1458
m=200	4.061	-1.532	0.628	0.398	96.204	55
m=500	4.073	-1.549	0.616	0.403	96.138	373
m=1000	4.065	-1.546	0.615	0.403	96.141	151

I used R this time.

c).

This time, I set learn rate $\alpha = 0.1$ and $m = 200$, which are acquired from best models in previous part. This table shows the comparison of the different methods.

	theta0	theta1	theta2	theta3	SSE	n.iter
SGD	4.061	-1.532	0.628	0.398	96.204	55
SGD with momentum(0.1)	4.064	-1.545	0.614	0.398	96.302	104
SGD with momentum(0.5)	4.065	-1.557	0.617	0.405	96.154	118
SGD with momentum(0.9)	4.239	-1.609	0.669	0.427	99.199	632
AdaSGD	4.065	-1.552	0.624	0.412	96.502	112

We can see that based on SSE, the SGD with momentum and AdaGrad algorithm do not perform much better than regular stochastic gradient descent, while the number of iterations of the two methods are at least twice more than that of SGD. So for this problem, SGD algorithm may be better.

Problem 2

a).

```
n <- 500
p <- 3
theta <- matrix(c(0.3, 0.6, -0.3, 0.2))
x <- list()
pi <- list()
y <- list()

logis <- function(x){
        return(1/(1+exp(-x)))
}

for (i in 1:5){
        x[[i]] <- cbind(rep(1, n), matrix(rnorm(n*p), nrow = n, ncol = p))
        pi[[i]] <- logis(x[[i]] %*% theta)
        y[[i]] <- round(pi[[i]])
}
percent_one <- sapply(y, sum)/(n*p)
percent_one
```

```
## [1] 0.2206667 0.2106667 0.2146667 0.2226667 0.2253333
```

b).

	theta0	theta1	theta2	theta3	cross-entroy	n.iter
(0.4, 0.45, -0.35, 0.25)	2.259	4.339	-2.259	1.407	0.115	220
(20, 20, 20, 20)	13.844	26.345	-12.287	9.321	0.032	668
(-20, -20, -20, -20)	8.382	16.452	-9.870	4.221	0.071	709
(0, 0, 0, 0)	2.297	4.468	-2.324	1.515	0.112	245
(0.3, 0.6, -0.3, 0.2)	2.443	4.681	-2.436	1.584	0.107	275

	theta0	theta1	theta2	theta3	cross-entroy	n.iter
alpha=0.1	1.450	2.588	-1.379	0.855	0.178	310
alpha=0.2	1.783	3.359	-1.733	1.136	0.144	287
alpha=0.3	2.091	3.999	-2.094	1.356	0.123	303
alpha=0.4	2.034	3.752	-1.972	1.267	0.130	197
alpha=0.5	2.264	4.268	-2.297	1.441	0.116	218
alpha=0.6	2.284	4.405	-2.314	1.527	0.113	196
alpha=0.7	2.568	4.864	-2.557	1.632	0.103	220
alpha=0.8	2.445	4.727	-2.394	1.567	0.107	173
alpha=0.9	2.472	4.831	-2.548	1.585	0.104	166

	theta0	theta1	theta2	theta3	cross-entroy	n.iter
m=10	1.459	2.230	-1.187	1.005	0.198	25

	theta0	theta1	theta2	theta3	cross-entroy	n.iter
m=100	2.653	5.108	-2.592	1.711	0.099	190
m=200	2.764	5.403	-2.813	1.802	0.094	221
m=500	3.255	6.363	-3.246	2.115	0.081	344

From the tables above, I can find that:

1. The algorithm is very sensitive to the starting values. To be specific, when the starting values are near origin, the algorithm will converge to the points which is also near origin (different points of convergence from different starting values are very close), but these results have a relative large error; when the starting values are far from the origin, the algorithm will converge to the points which is also far from origin (There points are also far from each other), but these results have a relative small error. It means that the objection function of this problem may not be convex and have multiple local minima far from zero.

2. The algorithm is not that sensitive to learning rate and batch size compared to starting values. The trend is that larger learning rate and larger batch size are, the smaller error is.

3. If we choose the wrong starting value, like origin, the algorithm cannot find the local minima far from the starting point, no matter what the learning rate and the batch size are, because it got stuck in the nearest local minima. So the choice of starting values is very important for SGD in this problem.

c).

I run the 10 times RMSProp with momentum under the same condition, and the results are listed below.

#	theta0	theta1	theta2	theta3	cross-entroy	n.iter
1	42.494	87.002	-43.078	28.944	0.003	632
2	21.100	43.572	-21.550	13.023	0.011	154
3	33.585	72.675	-35.869	23.157	0.008	410
4	31.810	57.449	-30.074	20.543	0.015	308
5	55.573	114.453	-57.462	36.449	0.002	1267
6	43.959	88.352	-44.850	29.799	0.002	742
7	26.704	51.994	-26.069	16.412	0.008	240
8	22.835	44.326	-24.008	14.047	0.013	186
9	54.773	112.962	-56.625	37.554	0.002	1332
10	44.314	90.795	-44.528	29.194	0.004	685

Generally speaking, the this algorithm performs much better than SGD. What is interesting thing, I think, is that RMSProp with momentum would not get stuck in local minima, which indicates that this algorithm is more robust. So the results of convergence is the points far from origin even the starting values is near the origin as before.