Big Regression - Challenge 1

Kwa Jie Hao

11/19/2017

Note: To save the reader's time, I have included only the essential details of this project in the main body of this document. For more details, I have logged carefully my progress for this assignment in the appendix and the attached code chunk R file. Sensitivity checks were carried out subject to computational limitations.

(1) Study the vignette of the sgd R package3 and familiarize yourself with the options the sgd function provides (particularly method, npasses, lr, and lr.control).

I have provided a summary in the appendix.

(2) sgd does not currently provide support for ffdf objects. Devise and describe a limited memory algorithm that uses the sgd function and carries out an explicit sgd update for logistic regression models.

For a limited memory algorithm, it was key that the large ffdf object not be called into memory. This meant calling it into memory in chunks. Initially, I thought of creating a small function that allowed me to access different rows of the ffdf object by outputing a range index. After some digging around the ff documentation, I found a function chunk, or chunk.ffdf, which does exactly that. By looping over the chunks, I ensure that only that particular chunk of the ffdf is called into memory. This is my algorithm:

```
sgd_chunk <- function(x, chunk_size=NULL, formula, sgd_model="glm",
                      sgd_modelcontrol=list(family=binomial(logit)), sgd_method="sgd",
                      lrvalue="adagrad", ...) {

  # if no chunk size is chosen, we let the chunk function decide based on RAM availability
  higgs_chunks <- chunk.ffdf(x, from=1, to=nrow(x), by=chunk_size)

  # we loop through the chunks and apply the sgd function to it
  for (i in higgs_chunks) {
    sgd.theta <-sgd(formula, data = x[i,], model = "glm", model.control=sgd_modelcontrol,
                    sgd.control = list(method=sgd_method, lr=lrvalue, ...))
  }

return(sgd.theta)

}
```

To choose the type of learning rate to use, I first computed the estimates I got from "adagrad", "one-dim" (the default), and "rmsprop" for all 3 Higgs formulae and compared them with the results obtained with bigglm, which computes a numerically stable solution of the MLE using incremental QR decomposition. I ignore "d-dim" and "one-dim-eigen" as I felt that "adagrad and"rmsprop" are more sophisticated models. I have attached the results in 3 dataframes for reference. The variance in results is not surprising since it is widely known that the performance of the explicit sgd algorithm is highly dependent on the tuning of γ in the learning rate used, other hyperparameters used with the learning rate. The results obtained with "adagrad" were most similar to the ones obtained with bigglm so I chose "adagrad" as the default option in my code.

We should expect that the answer obtained when using chunks to be consistent and the same as the answer when calling the whole ffdf object into memory with the sgd function. However, I noticed that my code was giving me different estimates depending on what chunk size I submitted. None of the results I obtained exactly matched the output obtained from in-memory calculation although 3 of them were close.

I first run the sgd_chunk function with traindat_ffdf (10.5 million observations) and set "adagrad" to be the value for lr. I obtained different values when allowing the computer to choose the chunk size, a chunk size of

1E6, and a chunk size of 3E6. When I let the computer choose the chunk size based on memory, it split the traindat_ffdf object into 151 chunks but the algorithm did not converge. However, the algorithm converged when using a chunk size of 1E6 but **did not** when using a chunk size of 3E6. To investigate further, I used a chunk size of 5E5, which splits traindat_fdff cleanly into 21 chunks, but again, it did not converge.

npasses

I suspected that there was something wrong with the way the chunks were being treated by the function. The default number of passes in the sgd_function was 3. This was a problem because it meant that the sgd function looped over each chunk 3 times before moving to the next chunk. To get my sgd_chunk function working like the origina, I revised my algorithm so that sgd_chunk only loops once all the chunks have been read. Another issue is shuffling. In this case, the shuffle option in the sgd package will only shuffle within chunks and not between chunks. For true shuffling of samples, the ffdf object to be submitted needs to be shuffled before being passed into sgd_chunk as an argument.

```
sgd_chunk_revised <- function(x, chunk_size=NULL, formula, sgd_model="glm",
sgd_modelcontrol=list(family=binomial(logit)), sgd_method="sgd", lrvalue="adagrad",
no_passes=3,...) {

  # if no chunk size is chosen, we let the chunk function decide based on RAM availability
  higgs_chunks <- chunk.ffdf(x, from=1, to=nrow(x), by=chunk_size)

  # we loop through the chunks and apply the sgd function to it
  for (j in 1:no_passes) {
    for (i in higgs_chunks) {
      sgd.theta <-sgd(formula, data = x[i,], model = "glm", model.control=sgd_modelcontrol,
                      sgd.control = list(method=sgd_method, lr=lrvalue, npasses=1, ...))
    }
  }

  return(sgd.theta)

}
```

learning rate

However, when comparing the results after making this change to my code, there was not much change. The algorithm again converged when the chunk size was 1E6 but not for the other chunk sizes. More importantly, the results obtained still did not match exactly the results obtained when calling the whole dataset into memory with sgd. This tipped me off that the issue lay with the learning rate - if the learning rate was passed on continuously from chunk to chunk, the estimates obtained should be identical regardless of chunk size; the learning rate resets everytime the function is called. It is interesting to note that when calling the entire traindat_ffdf object into memory and using it as an input into the sgd function, convergence is not achieved, but when a chunk size of 1E6 is used convergence is achieved. This hints that it is a combination of a certain value of learning rate together with a specific sequence of data points which causes the results between function calls with different chunk sizes to diverge. My guess is that the combination of a sequence of data points and the learning rate values caused it to converge at a local minimum. Running the data through more passes does not change the results substantially: they still do not converge.

My goal was to create a function so that the results obtained are identical to the results obtained from calling the whole dataset into memory with sgd. However, that would only be possible if I were able to somehow pass the learning rate from a previous call to a new call of sgd. Given that the sgd function did not allow us to specify our own learning rates, but only to choose from the options that were included (users are only allowed to set values for the hyperparameters for the included learning rate options), I was not able to find a way to chunk the data for use with sgd while preserving information on the learning rate, especially since the package was mostly written in C++, of which I have no knowledge of. Thereafter, I will be using the values I obtained from my chunking sgd function with the following specification:

```
mod_sgd1 <- sgd_chunk_revised(traindat_ffdf, formula=HiggsFormula1, lr="adagrad")
mod_sgd2 <- sgd_chunk_revised(traindat_ffdf, formula=HiggsFormula2, lr="adagrad")
mod_sgd3 <- sgd_chunk_revised(traindat_ffdf, formula=HiggsFormula3, lr="adagrad")
```

I allow the computer to choose the chunk size, and use the default values for "adagrad".

alternative solutions

An alternative which I thought of was streaming data into the sgd function (refer to appendix and code chunks). Ideally, the data stream could stream the data into the sgd function while it was running. However, I realized later that the get_points feature which streaming packages use create a dataframe in-memory which might be prohibitive in size if we were to get all the data points at once. Thus, any usage of streaming packages will require multiple calls of the sgd function, resulting in the same issue of not being able to transfer information on learning rate to the next call of sgd. An ideal solution would be an update function for the sgd package and storing learning rate information in the output sgd object.

(3) Implement your algorithm and use it to estimate the logistic regression models with formulae HiggsFormula1, HiggsFormula2, HiggsFormula3 from the 10.5 million observations in traindat ffdf (see slides), and carry out the out-of-sample ROC analysis using testdat ffdf as we did with biglm. Compare with the biglm fits and report your findings.

To obtain the predicted values, again, it is important to do it in chunks so that only limited memory is used, and also to save the output in the ffdf object which is stored on the hard drive. I have listed a sample for how I obtained the fitted values for sgd for HiggsFormula1 below. The rest are in the code chunks.

```
testdat_ffdf$fitted4 <- as.ff(as.numeric(NA), length = nrow(testdat_ffdf))
testdat <- data.matrix(as.data.frame(testdat_ffdf))
chunks <- chunk.ffdf(testdat_ffdf)

for(chunkrangeindex in chunks){
  testdat_ffdf$fitted4[chunkrangeindex,] <-
    predict(mod_sgd1, cbind(1,data.matrix(as.data.frame(testdat_ffdf[chunkrangeindex, ]))
                            [, all.vars(HiggsFormula1[[3]])]), type = "response")[,1]
}
```

Then I use the same ROC function found in the slides to produce ROC curves for the fitted values using the sgd package. The figures, in fact, look very similar in the joint plot of the curves in Figure 1 at the end of this section (the rest of the plots can be generated from the code chunks):

The curves are very close for Higgs Formulae 1 and 2. This is not surprising given the similarity in parameter estimates obtained. The most notable difference is found for Higgs Formula 3, where the curve for the sgd model is hovering around the y=x line, indicating poor accuracy, whereas the curve for the bigglm model has a large looping arc (Figure 2), indicating greater accuracy. The rate of true positives and false positives is approximately equal for sgd in this case.

I've also computed the ROC curves for the estimates obtained with "rmsprop" (Figure 4) and "one-dim" (Figure 3) and compared them with "adagrad" (Figure 5), the option I chose. All of the "rmsprop" curves have poor accuracy, but the difference is closer between "one-dim" and "adagrad".

Comparing "adagrad" and "one-dim", "one-dim" performs better on average, but "adagrad" performs the best with HiggsFormula2. I have also included a ROC plot comparing "one-dim" with the the bigglm values (Figure 6).

Overall, the choice of "adagrad" over the other learning rate options is not a bad option, but it is still unclear what caused the poor performance of "adagrad" in HiggsFormula3.

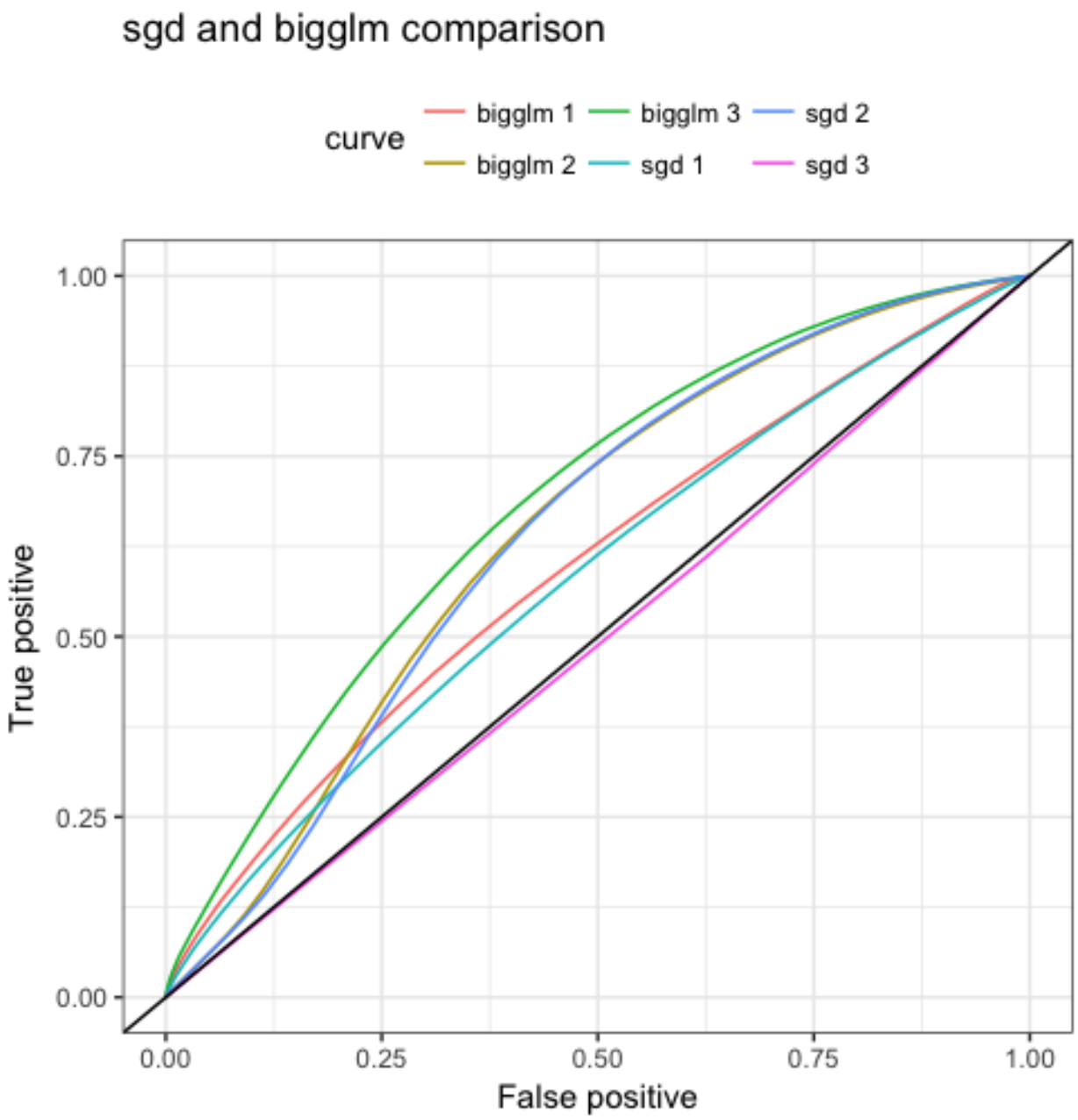
sgd and bigglm comparison
curve
bigglm 1
bigglm 3
sgd 2
bigglm 2
sgd 1
sgd 3
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 1:

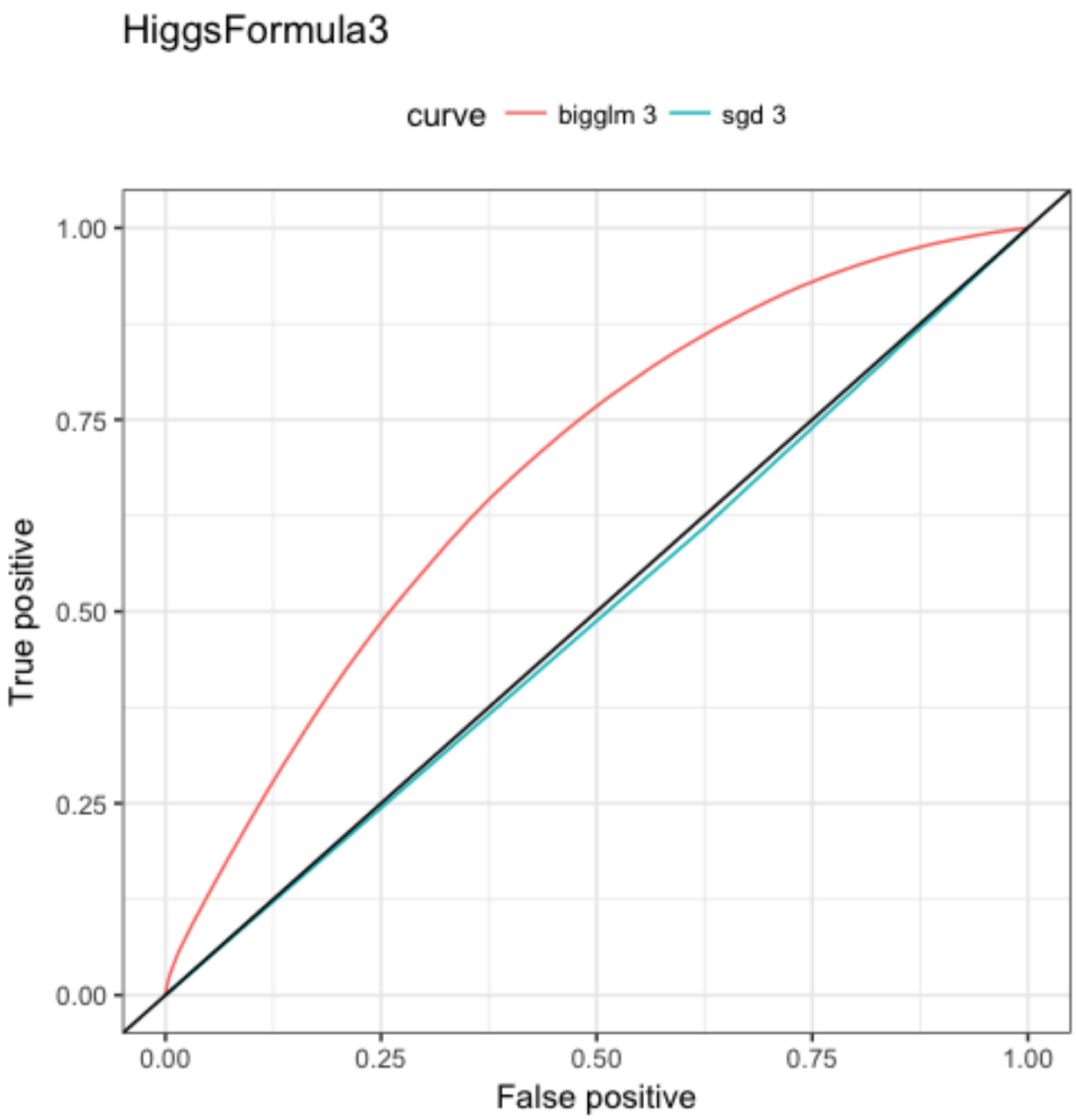
HiggsFormula3
curve
bigglm 3
sgd 3
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 2:

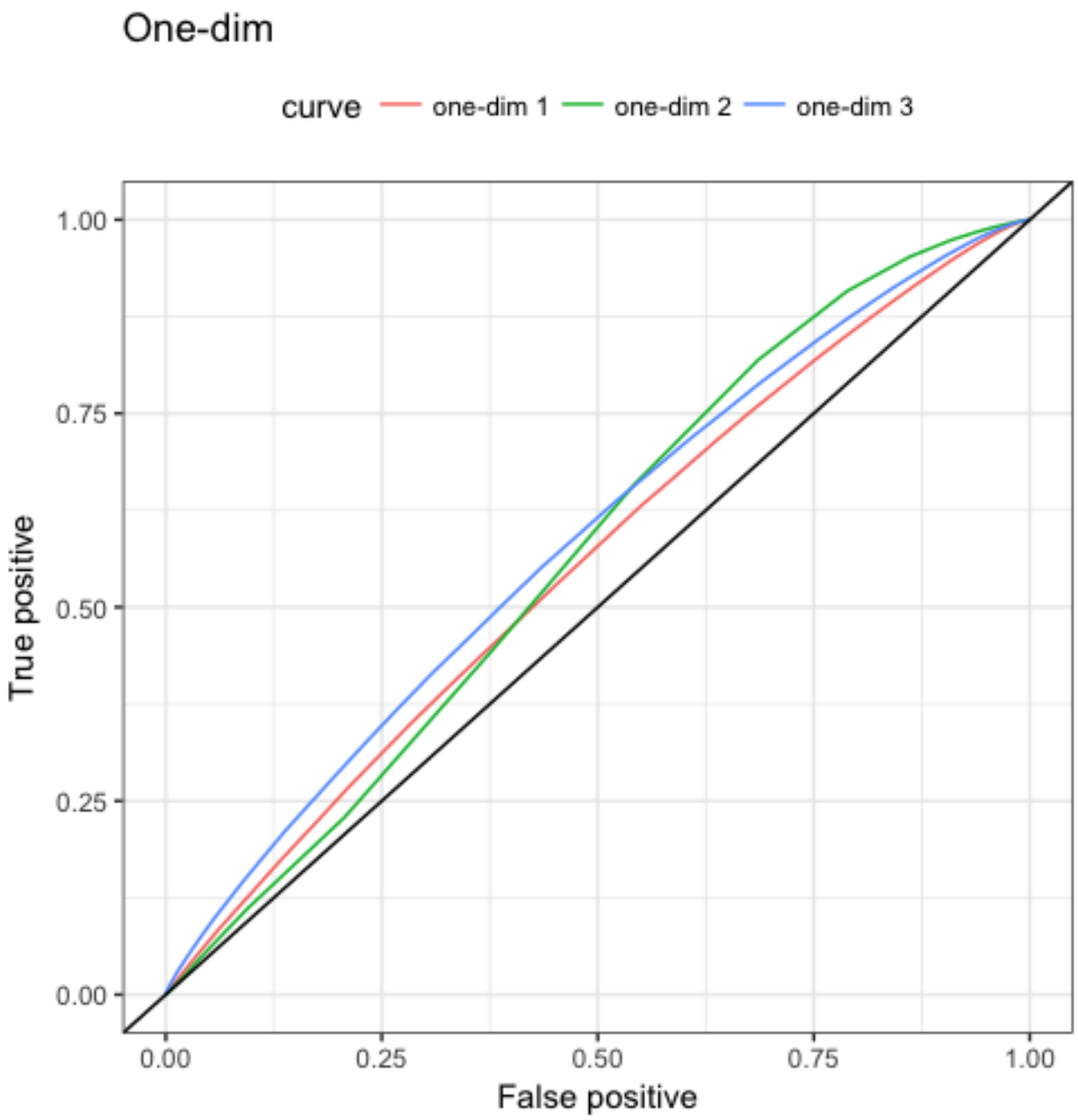
One-dim
curve
one-dim 1
one-dim 2
one-dim 3
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 3:

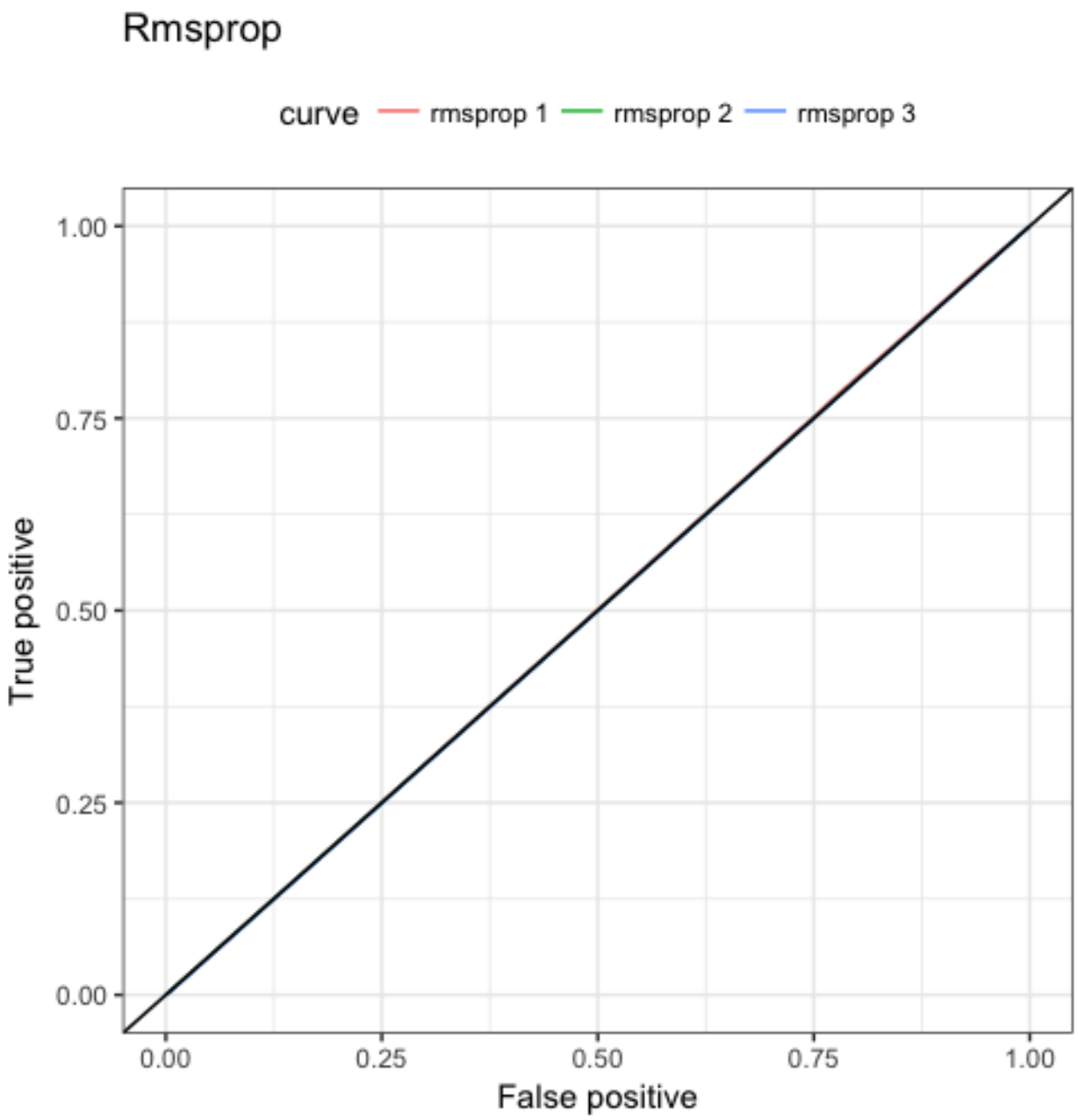
Rmsprop
curve
rmsprop 1
rmsprop 2
rmsprop 3
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 4:

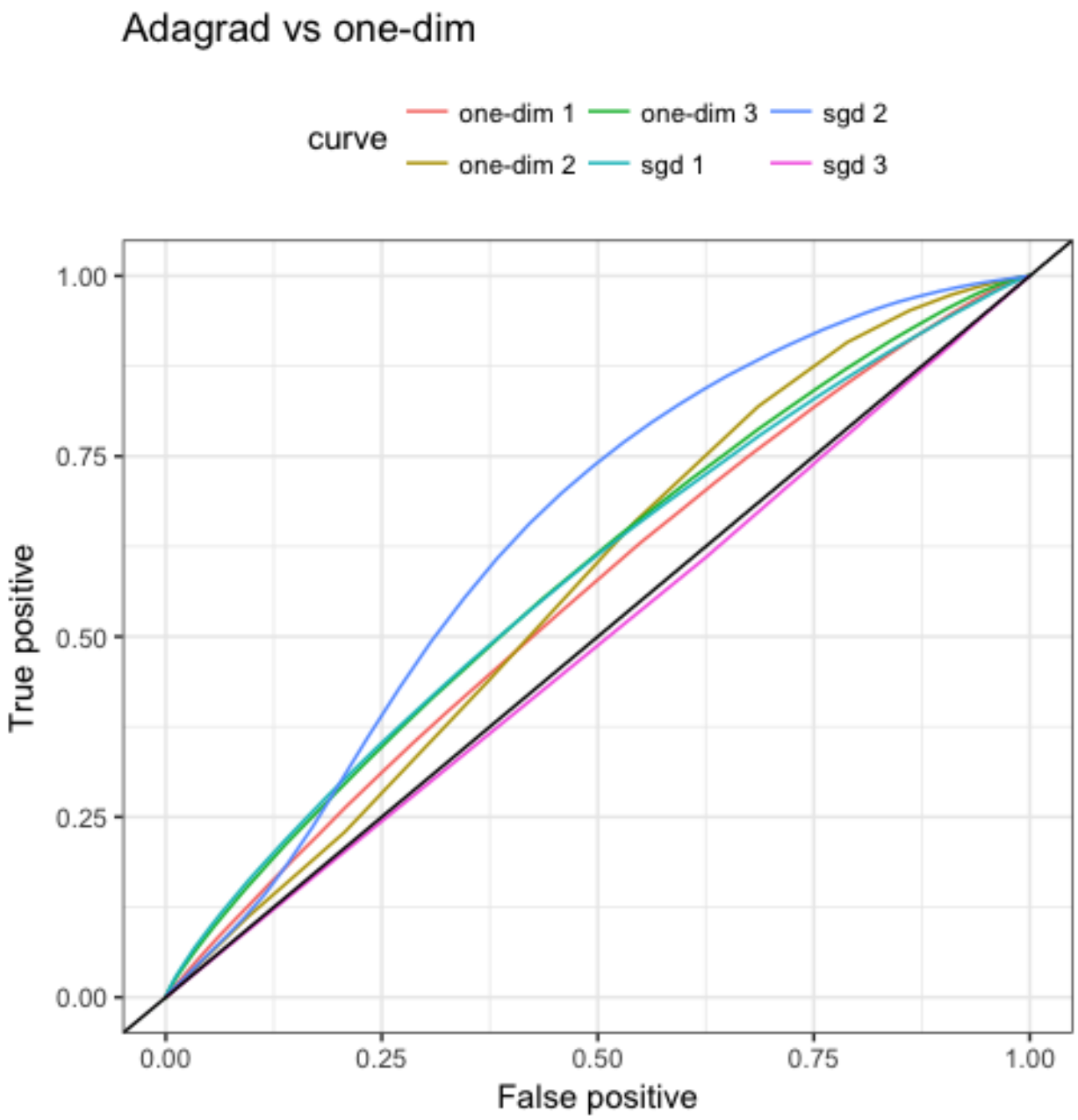
Adagrad vs one-dim
curve
one-dim 1
one-dim 3
sgd 2
one-dim 2
sgd 1
sgd 3
True positive
False positive
0.00
0.25
0.50
0.75
1.00

Figure 5:

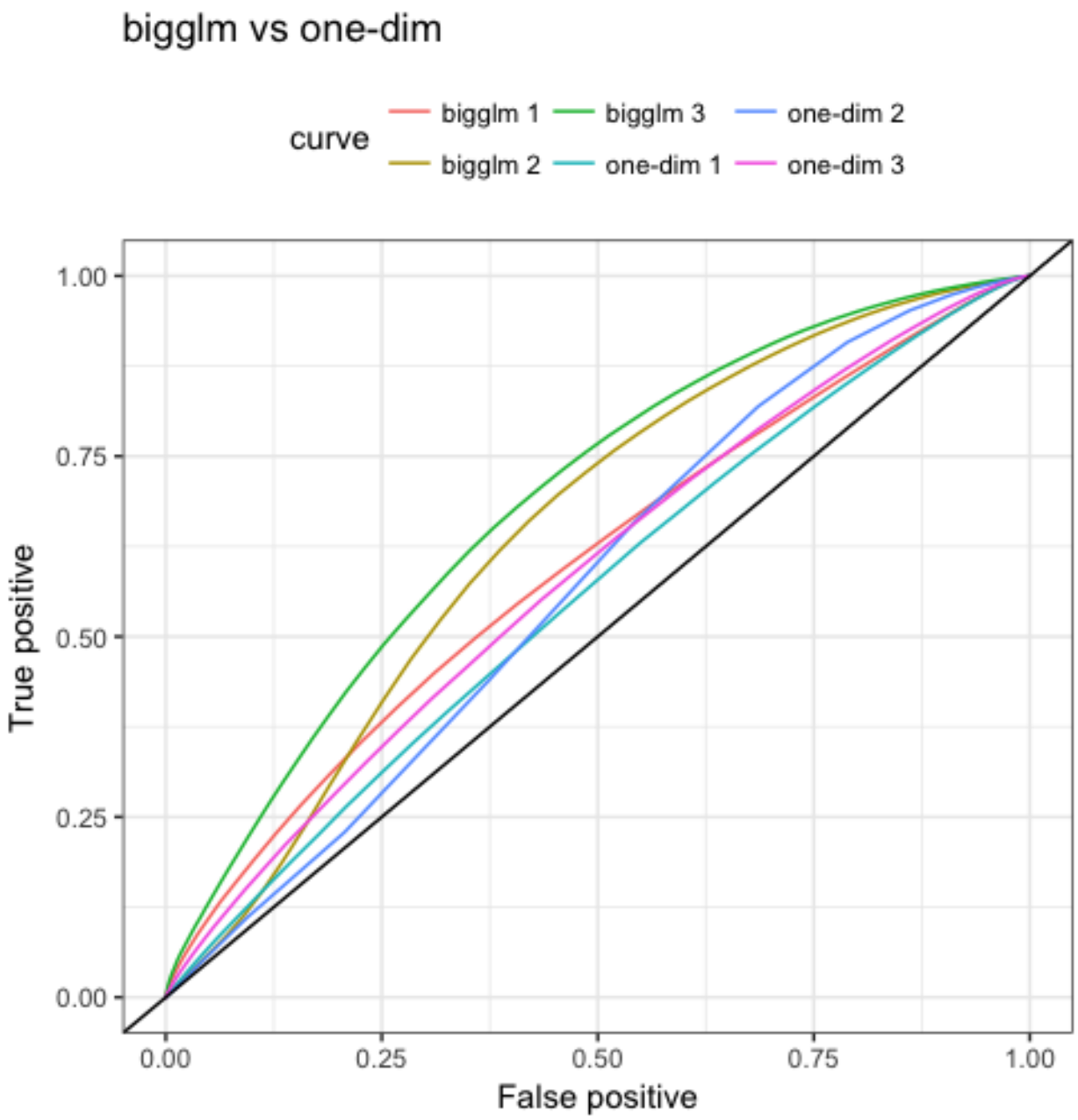
bigglm vs one-dim
curve
bigglm 1
bigglm 3
one-dim 2
bigglm 2
one-dim 1
one-dim 3
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 6:

(4) Carry out sensitivity checks on your solutions and comment on the results.

In this section, I analyze the different sources of uncertainty which contributes to possible variation in my results. In this context, this refers to the choice of learning rate, values of hyperparameters for learning rates, the formula used, the starting value of the stochastic gradient descent, whether the data is shuffled or not and the number of passes over the data.

Here, I choose to conduct sensitivity checks using the whole training subset as opposed to a subset of the training subset that was used in the slides. This is because I do not know how to shuffle the ffdf object and there might be patterns in the data by index.

I have already shown throughout this document that the choice of learning rate greatly affects the results obtained by sgd. The toy example on slide 106 of the Big Regression presentation helps to illustrate why explicit sgd estimates fluctuate substantially when learning rate parameters are changed. When the value of gamma is changed to $1/i$ as opposed to the original of $100/i$, the amplitude of the graph becomes very small and the estimate hovers around the MLE. On the contrary, when gamma is set to $1000/i$, the amplitude becomes very large and the sgd estimate fluctuates wildly. This is because the learning rate is too high and bounces around the optimum value since the steps taken by the algorithm are large. This is made even worse when gamma is set to $100/sqrt(i)$; the estimate fluctuates wildly without converging. When gamma is set to $100/i^2$, the estimate converges to a different value to the MLE. Thus, when using "adagrad", "one-dim" and "rmsprop" which all have different gammas, the estimates obtained are inconsistent and different in magnitude and fluctuation patterns.

starting values

We have also seen how the formula used, and thus the predictors used, affects the estimates (relative to a certain standard). Compared to bigglm, sgd performs admirably using "adagrad" but there is a huge drop in performance when it reaches HiggsFormula3. I use the case of HiggsFormula3 and compare different starting values on a ROC plot. I already have the ROC values for the sgd starting from zero (mod_sgd3) and the MLE (mod_bigglm3). I computed the ROC values for starting from the MLE and from half of the MLE and plotted them in figure 7 (accompanying code in code chunks). [Note that there is no point in checking sensitivity on starting values for the other two formulae since the sgd function comes up with estimates very close to the MLE even when it starts from zero]

Despite starting from the MLE value, the sgd function was not able to arrive at the MLE value and instead converged on a different value. The sgd function which started from half the MLE value also converged, but also not on the MLE value. Studying the ROC plot, it seems that the closer the starting value to the MLE value, the more accurate the function is as it moves outward toward the top left corner, but the change in accuracy is minimal as the ROC curves are close together. It appears that a good starting value can help the sgd algorithm to increase its accuracy minimally, but it is not a huge improvement compared to starting from zero.

On the other hand, the difference in starting value led to convergence on different estimates, so in that sense, the starting value is important to obtaining better results.

I repeat this for lr="one-dim" in figure 8. The results support my conclusion above.

lr.control

"adagrad" with the default options performed terribly with HiggsFormula3. Is this because of some problem with "adagrad" or is it a problem with the learning rate hyperparameters? I generated four different samples from sgd_chunk_revised function using different hyperparameters for η, the constant in the conditioning matrix, using $\eta = 0.7, 0.9, 1.1, 1.3$ (quick checks showed that sensitivity to epsilon is minimal for minor changes in epsilon). The results are shown in figure 9: performance remains roughly the same but improve by a huge amount when $\eta = 0.7$ In fact, it comes close to performance with the MLE (figure 10).

I carry out checks with $\eta = 0.7$ for HiggsFormula1 and HiggsFormula2 to check for consistent performance. I also check for $\eta = 0.8, 0.71$ for HiggsFormula3 since the jump in performance is so large. I find that the sgd_chunk_revised function performs excellently when $\eta = 0.7$ (figure 11) as its roc curves are very close

to the bigglm roc curves. I also find that performance of sgd "adagrad" remains terrible at $\eta = 0.8$ but are almost the same for $\eta = 0.71, 0.7$ (figure 12).

The results confirm that hyperparameter settings for explicit sgd can be extremely sensitive and impact performance crucially. Performance changed drastically within a 0.09 interval, most likely even smaller than that. Poor performance of default hyperparameter values perhaps explains the poor performance of "rmsprop" in this assignment as well. Without the MLE values obtained from bigglm, it would be a challenge to obtain good values of *eta* to use.

npasses

The sgd function did not converge for HiggsFormula1 using the default value of three passes. I conduct sensitivity checks by number of passes over the data to see how much the estimates change. By observing the data frame "passes" included in the code chunks, the values are very, very close together - the change in estimates is small for every increase in the number of passes over the data and slowly approaches a certain value: in my case, the values do not approach the MLE because of the flawed sgd_chunk_revised function which is unable to pass on the learning rate from chunk to chunk.

The estimates are not very sensitive to the number of passes, although increasing the number of passes over the data will result in a convergence. However, this is not necessarily useful since the notion of convergence, or tolerance, can be defined by the user. Figure 13 shows that the points of the coefficients for different passes are very close to each other.

shuffles

As mentioned earlier, I was unable to shuffle the ffdf object while keeping it in memory as I was using the large training dataset. Furthermore, because of the way my sgd_chunk_revised function was specified, I was unable to use the shuffle function provided in the sgd function properly since it would only shuffle within chunks, which would reduce the efficacy of the stochastic aspect of sgd.

(5) Final Remarks

In hindsight, my choice to go with "adagrad" after comparing performances with "one-dim" and "rmsprop" was a naive one because explicit sgd is a highly sensitive procedure and dependent on a number of hyperparameters. "adagrad" happened to perform best with the default parameters, but any of the other learning rate options could have been tuned to perform as well or even better with the right parameter values.

Furthermore, it turns out that not being able to pass the learning rate from chunk to chunk does not affect the results as much as I thought it would. By choosing a certain learning rate and tuning the hyperparameters, I obtained solutions that were very close to the MLE. I'm sure I could have improved on those estimates and obtain solutions that are even more similar to the MLE with more tuning. It remains to be seen how shuffling the data would affect estimates, although I would expect it to increase efficacy as it ensures that the data is properly randomized.

For this project, we were fortunate enough to have the results from the bigglm function to compare with. In reality, cross-validation would have to be performed and perhaps the roc curves for many different hyperparameter settings compared. This will be a lot more computationally intensive than what I've done here.

Overall, I felt that I learnt a lot about the process and mathematics of stochastic gradient descent. However, in the process of my work I found out about a lot more details and nuance to stochastic gradient descent such as the various gradient descent optimization methods in addition to "adagrad" like "adam", "adamax", "adadelta" among much more. Despite being at the end of this assignment, I feel more like I've just begun.

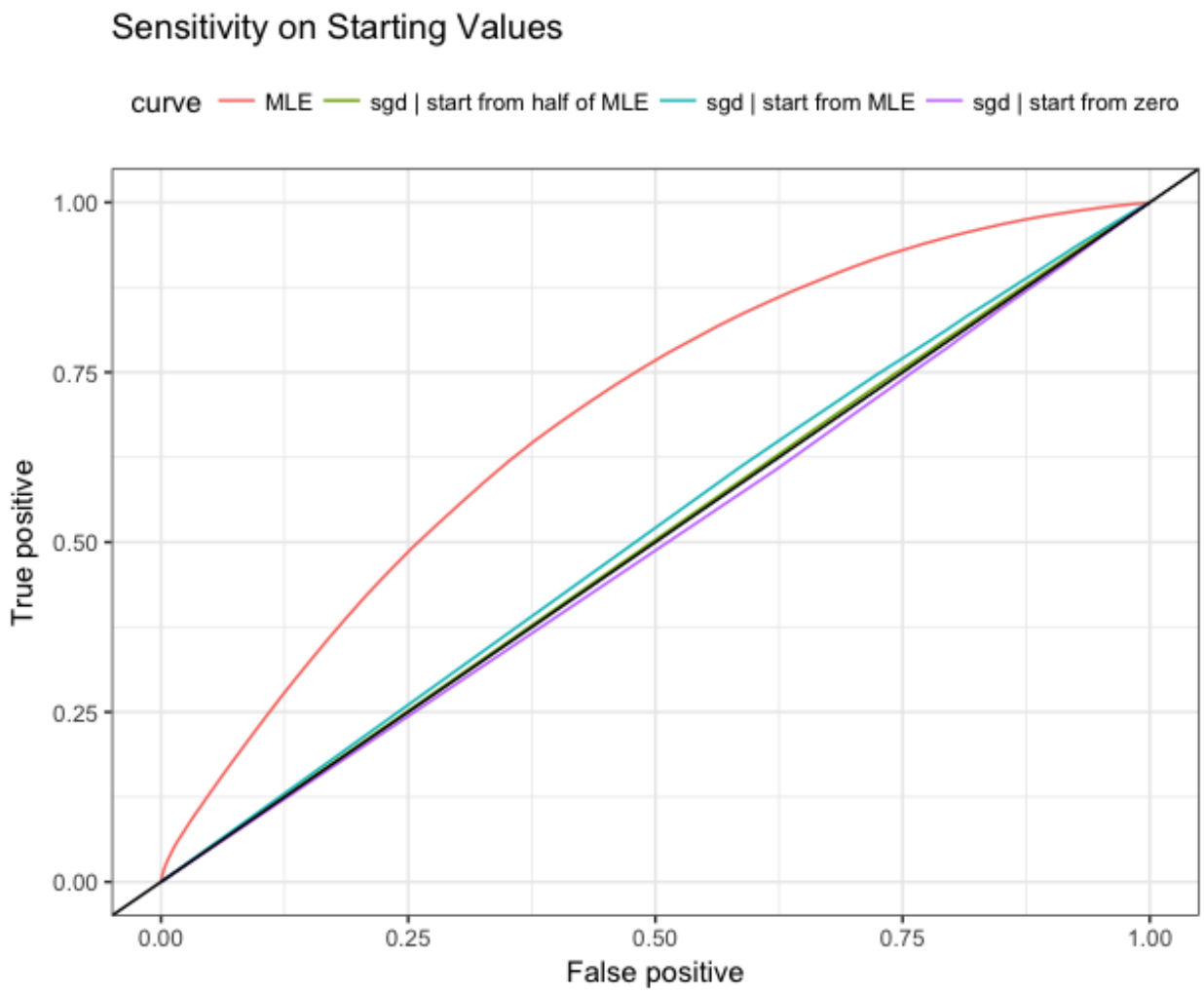
Sensitivity on Starting Values
curve
MLE
sgd | start from half of MLE
sgd | start from MLE
sgd | start from zero
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 7:

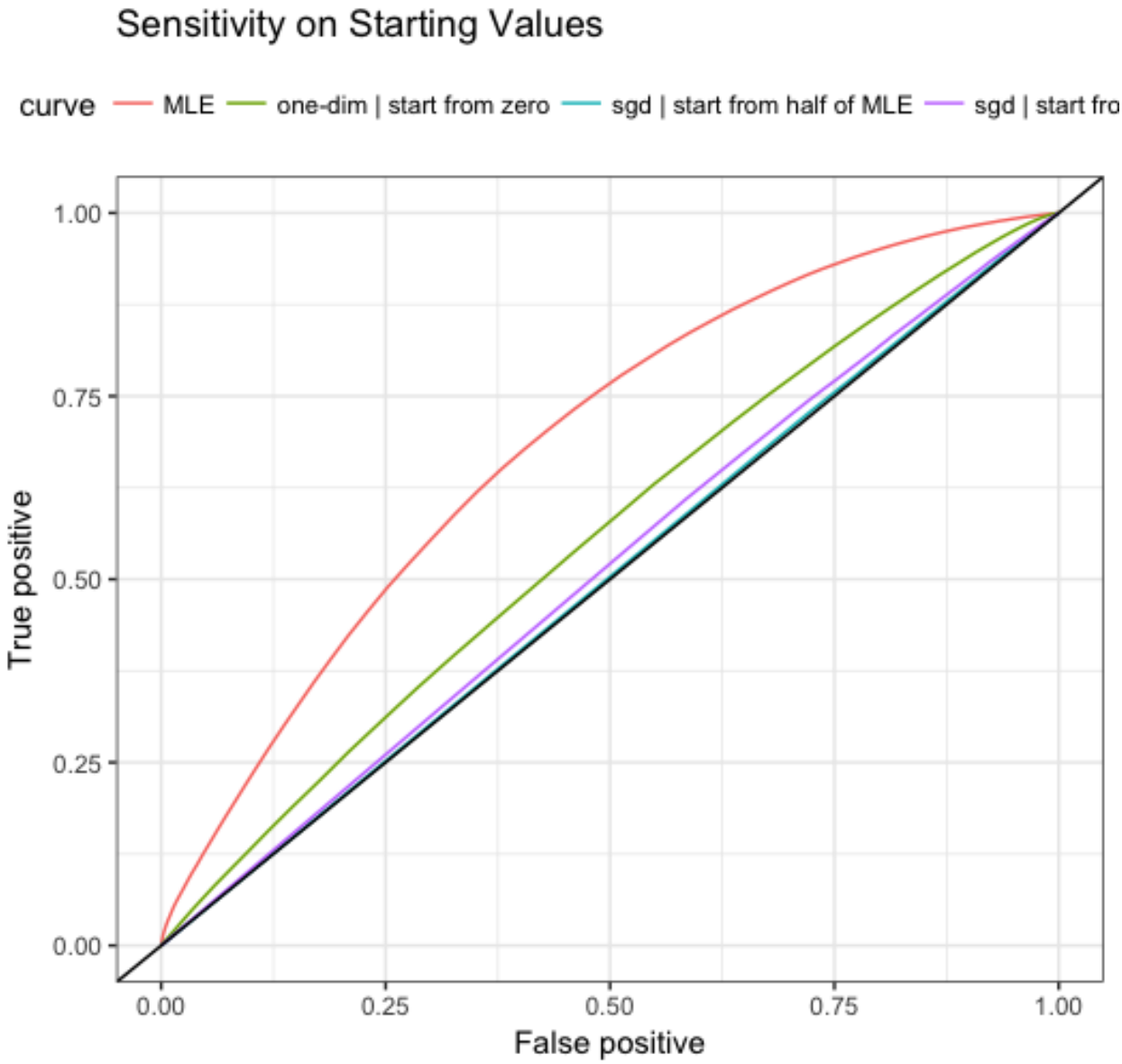
Sensitivity on Starting Values
curve
MLE
one-dim | start from zero
sgd | start from half of MLE
sgd | start fro
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 8:

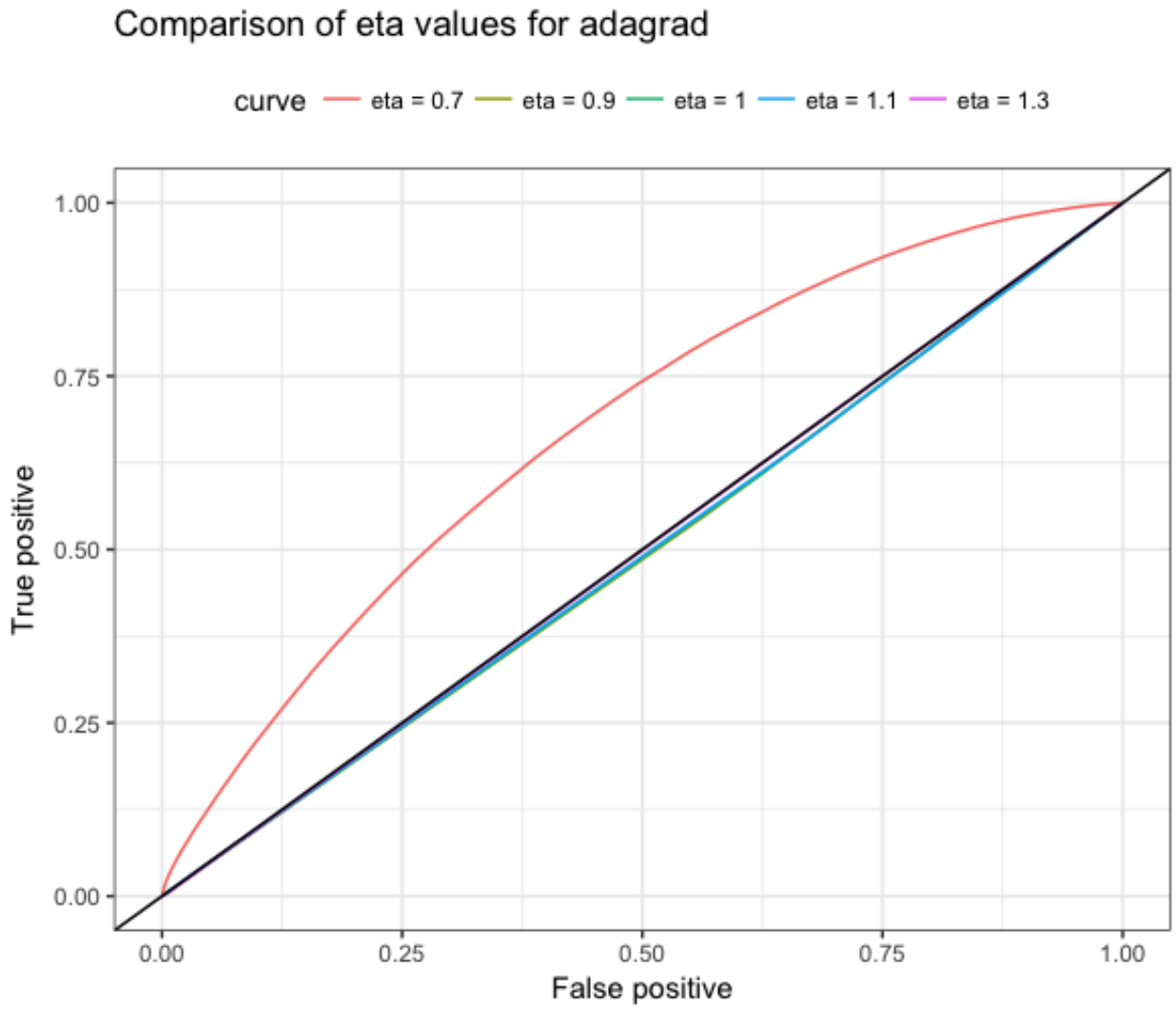
Comparison of eta values for adagrad
curve
eta = 0.7
eta = 0.9
eta = 1
eta = 1.1
eta = 1.3
True positive
False positive
0.00
0.25
0.50
0.75
1.00

Figure 9:

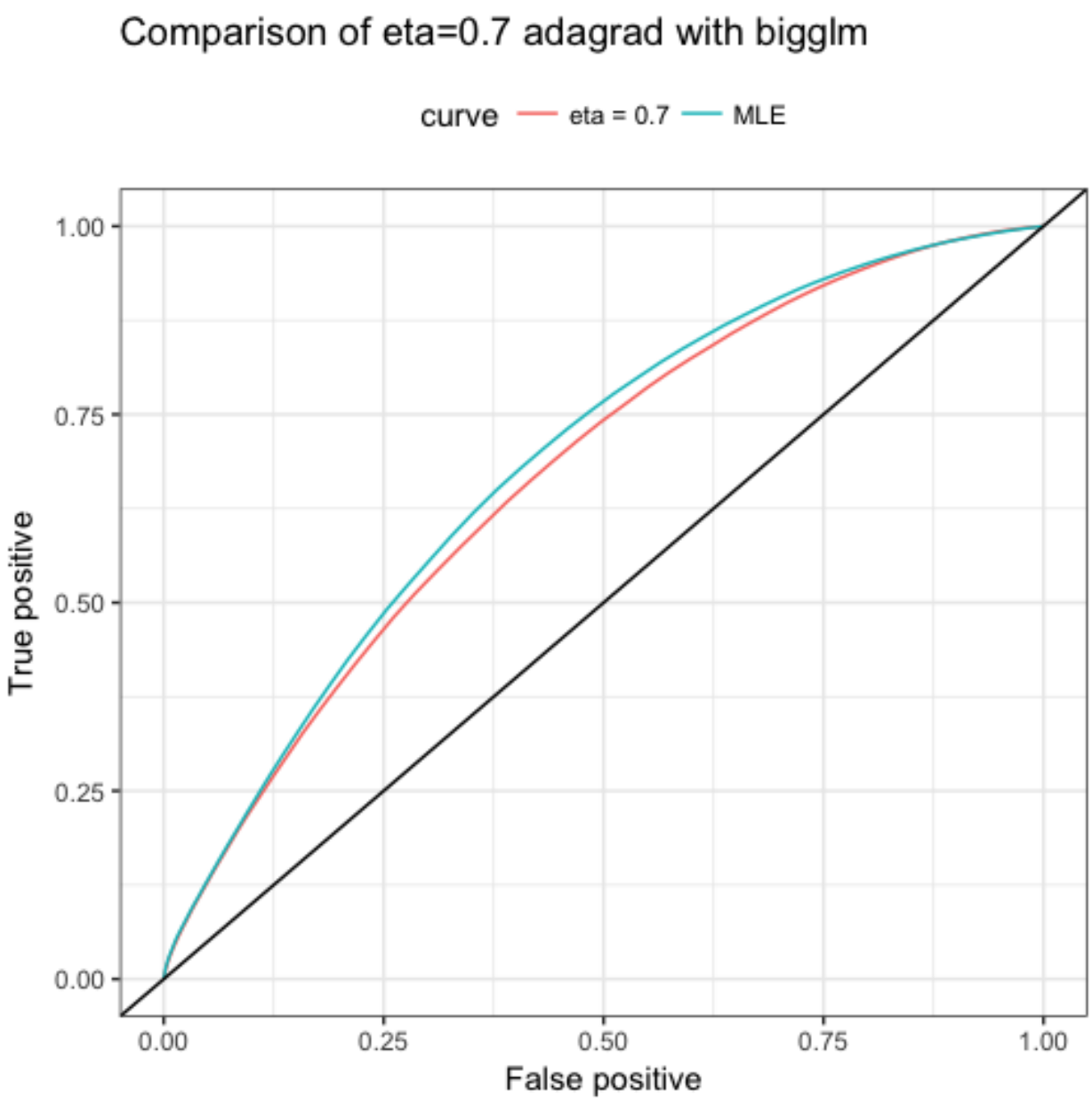
Comparison of eta=0.7 adagrad with bigglm
curve
eta = 0.7
MLE
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 10:

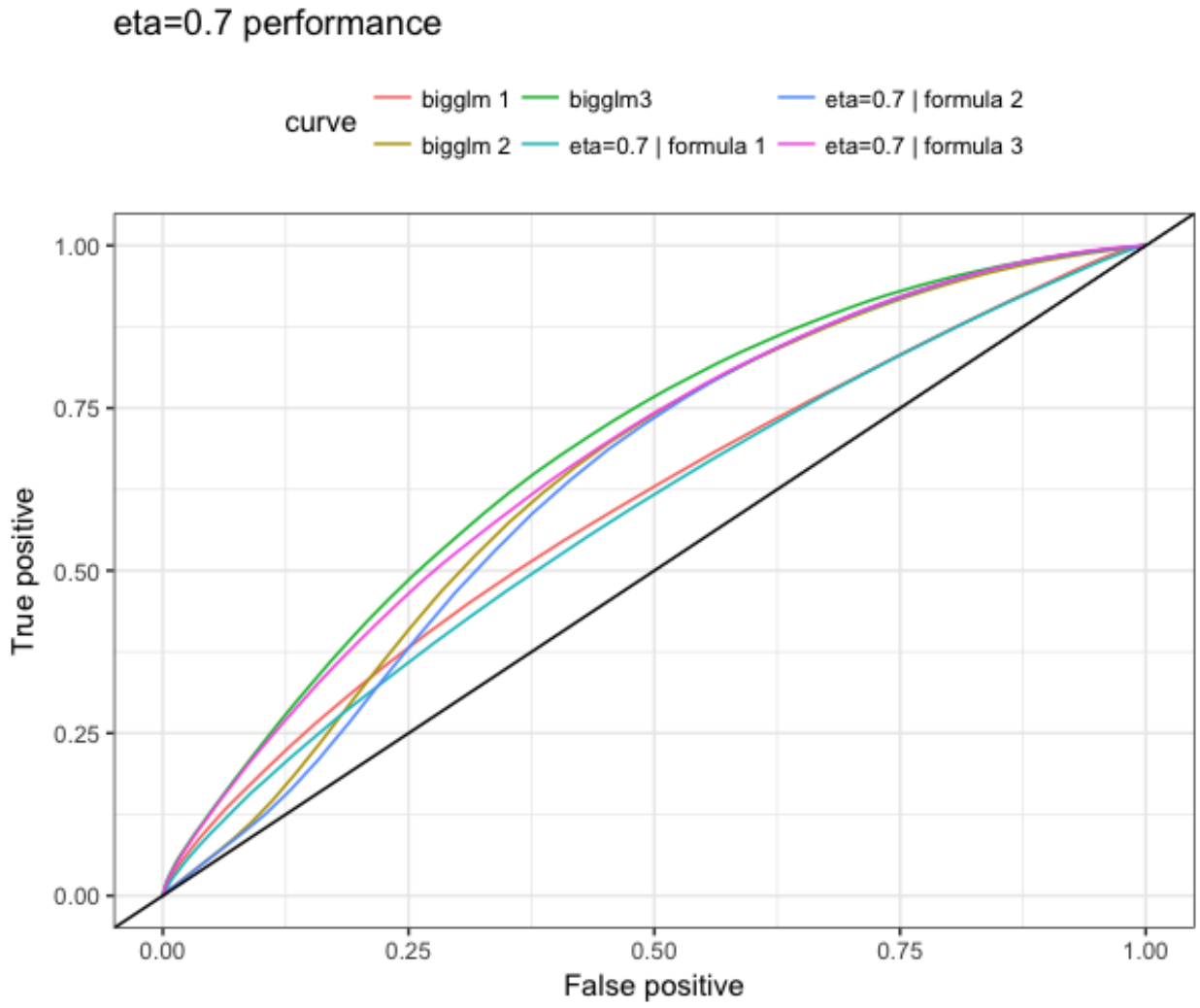
eta=0.7 performance
curve
bigglm 1
bigglm3
eta=0.7 | formula 2
bigglm 2
eta=0.7 | formula 1
eta=0.7 | formula 3
True positive
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00
False positive

Figure 11:

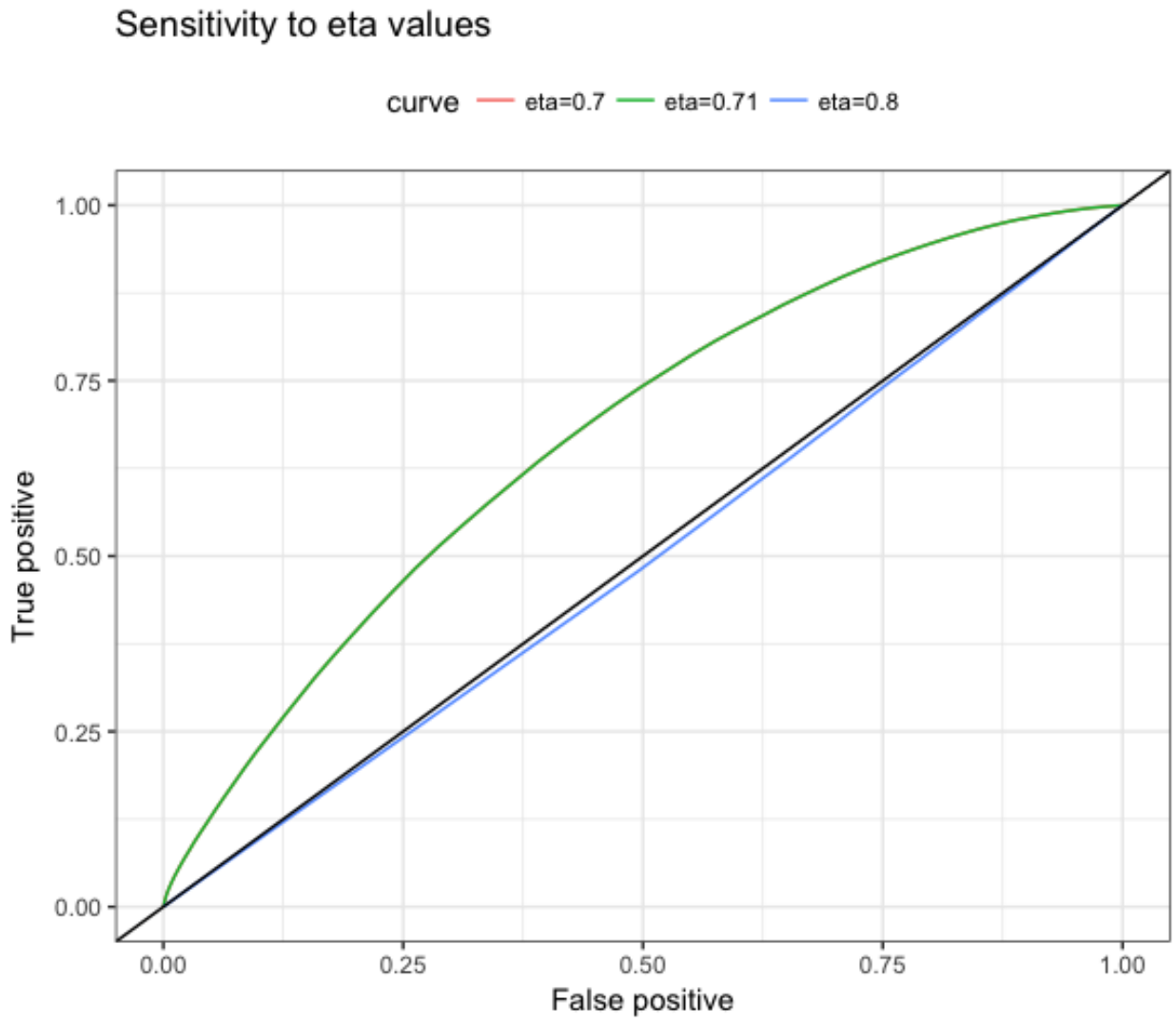
Sensitivity to eta values
curve
eta=0.7
eta=0.71
eta=0.8
1.00
0.75
0.50
0.25
0.00
True positive
0.00
0.25
0.50
0.75
1.00
False positive

Figure 12:

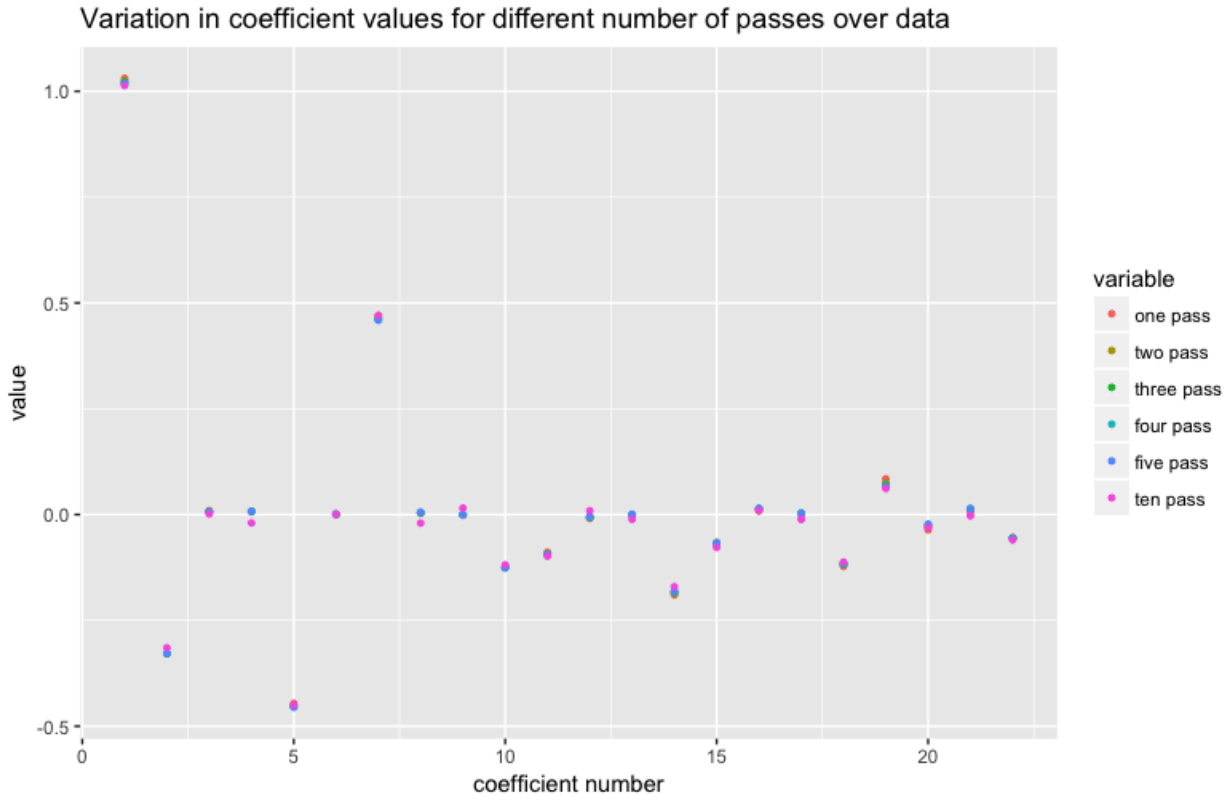
Variation in coefficient values for different number of passes over data
value
1.0
0.5
0.0
-0.5
0
5
10
15
20
coefficient number
variable
one pass
two pass
three pass
four pass
five pass
ten pass

Figure 13:

References

Big Regression Presentation Slides

Xu, W. (2011). Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.

Duchi, J., Hazan, E., & Singer, Y. (2017). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 2121-2159.

Tieleman, T. and Hinton, G. (2012) Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude. COURSERA: Neural Networks for Machine Learning, 4, 26-31.

Appendix

(1)

method describes the method to be used by the sgd function. This includes the original stochastic gradient descent, implicit stochastic gradient descent, stochastic gradient descent with averaging (ai-sgd) and others. The key equations are as follows:

explicit sgd: $\theta_n^{ex} = \theta_{n-1}^{ex} + \gamma_n C_n \nabla logf(y_n; x_n, \theta_{n-1}^{ex})$

implicit sgd: $\theta_n^{im} = \theta_{n-1}^{im} + \gamma_n C_n \nabla logf(y_n; x_n, \theta_n^{im})$

averaged implicit sgd (ai-sgd): $\theta_n^{sgd} = \theta_{n-1}^{sgd} + \gamma_n C_n \nabla logf(y_n; x_n, \theta_{n-1}^{sgd})$, and then $\bar{\theta}_n = (1/n) \sum_{i=1}^{n} \theta_i^{im}$

averaged sgd (asgd): similar to ai-sgd but for the explicit sgd

They also include older techniques.

classical momentum (CM) uses the update: $v_n = \mu v_{n-1} + a_n \nabla logf(y_n; x_n, \theta_{n-1})$, and $\theta_n = \theta_{n-1} + v_n$ where $\mu \epsilon [0, 1]$ is a fixed momentum coefficient

nesterov's accelerated gradient (NAG): $v_n = \mu v_{n-1} + a_n \nabla logf(y_n; x_n, \theta_{n-1} + \mu v_{n-1})$. This is almost akin to a implicit version of classical momentum, paralleling the situation between explicit and implicit sgd.

npasses is the maximum number of passes over the data. The default number of passes is 3.

lr represents the character specifying the learning rate to be used: "one-dim", "one-dim-eigen", "d-dim", "adagrad", "rmsprop".

lr.control represents the vector of scalar hyperparameters one can set dependent on the learning rate. For hyperparameters aimed to be left as default, specify NA in the corresponding entries. For example, From the sgd documentation:

"one-dim" is the scalar value where lr.control = (scale=1, gamma=1, alpha=1, c) where c is 1 if implemented without averaging, 2/3 if with averaging. Optimized for use with averaged sgd.

"one-dim-eigen" is a diagonal matrix with lr.control left as null

"d-dim" is a diagonal matrix with lr.control = (epsilon=1e-6)

adagrad is a diagonal matrix with lr.control = (eta=1, epsilon=1e-6) proposed by Duchi et al. (2011). It dynamically incorporates information from data observed in previous iterations to improve gradient-based learning in future iterations so that it can. It is especially useful when data is sparse.

rmsprop is a diagonal matrix with lr.control = (eta=1, gamma=0.9, epsilon=1e-6) proposed by Tieleman and Hinton (2012). Rmsprop uses the magnitudes of recent gradients to normalize the gradients found in each iteration. It is a robust optimizer suitable for mini-batch learning.

(2)

First, we load the possible required libraries and data for this challenge (refer to code chunks).

When calculating the in-memory values obtained by sgd, I choose to use "adagrad" for the setting of "lr", leaving the values of the hyperparameters to be default. "adagrad" is more sophisticated than "one-dim" and uses information obtained from previous iterations of the sgd algorithm to improve gradient learning for future iterations."rmsprop" is useful for mini-batch learning. I do not use a starting value and allow the sgd function to randomly initialize around zero. For now, I let the value of "npasses" remain at its default of 3. Ultimately however, I chose to use "adagrad" because it gave similar results to those obtained from bigglm, which is the MLE, which uses the numerically stable incremental QR decomposition to obtain its results, whereas "one-dim" and "rmsprop" a gave substantially different results. Refer to the attached dataframes "results1", "results2" and "results3" for comparisons. However, it is important to note that the sgd algorithm was not deemed to have converged when "adagrad" was used.

Since the "lr" parameter only accepted pre-determined options as arguments, this meant that instead of feeding updated parameters into the sgd function, I could instead feed data into the sgd function using a streaming function. I found the packages **RMOA** and **stream** which contained data streaming functions which allowed the user to control how many data points to stream from a ffdf object. The code is included in the code chunks file.