Homework 5 and 6
APPM 5630 Spring 2023
Advanced Convex Optimization

Due date: Friday, Mar 3 2023 at midnight (via Gradescope) **Instructor**: Prof. Becker
Theme: Convex optimization **Revision date**: 2/24/23

Instructions Collaboration with your fellow students is allowed and in fact recommended, although direct copying is not allowed. The internet is allowed for basic tasks. Please write down the names of the students that you worked with. An arbitrary subset of these questions will be graded.

Homework submission instructions at github.com/stephenbeckr/convex-optimization-class/tree/master/Homeworks. You'll turn in a PDF (either scanned handwritten work, or typed, or a combination of both) to **gradescope**, using the link from the Canvas assignment.

Reading Read chapter 4 in [BV2004]. We will skip sections 4.5 and 4.7 in lecture (geometric programming and vector optimization), so please at least skim these to get an idea of the subject.

Homework 5

Background Both Homework 5 and Homework 6 relate to **logistic regression**. We follow Kevin Murphy's *Machine Learning* (2012). On our canvas website, we have included two pages of introduction to logistic regression from his textbook, so we recommend you read that first.

We follow machine learning notation, and let $\mathbf{w} \in \mathbb{R}^p$ be our decision variable, which represents weights for the columns of the $n \times p$ data matrix X, where the i^{th} row of X is $\mathbf{x}_i^T$. We use bold for vectors, while scalars and matrices are not bold (but matrices are capital letters). Let

$$\sigma(a) \overset{\text{def}}{=} \frac{1}{1+e^{-a}} = \frac{e^a}{e^a+1}$$

be the **sigmoid/logistic/logit** function. See Fig. 1 left. This function is not convex, but $-\log(\sigma(a))$ is convex, and this is what is used in maximum likelihood.

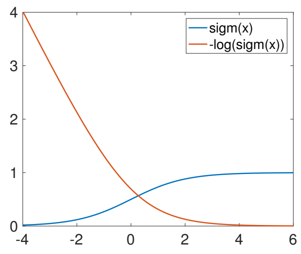

Figure 1: The sigmoid function σ.

We assume that we collect data $\mathbf{x}_i$, and a binary response variable y_i which is the **label**, where

$$y_i = \begin{cases} +1 & \text{with probability } \sigma(\mathbf{w}^T\mathbf{x}_i) \\ -1 & \text{with probability } 1-\sigma(\mathbf{w}^T\mathbf{x}_i) \end{cases}.$$

We can write the above in the compact form $p(y_i \mid \mathbf{w}, \mathbf{x}_i) = \sigma(y_i \mathbf{w}^T \mathbf{x}_i)$ since $\sigma(a) = 1 - \sigma(-a)$. If we collect $\mathbf{y} = (y_i)_{i=1}^n$ observations, then the probability of observing this realization of the random variable is $p(\mathbf{y} \mid \mathbf{w}, X) = \prod_{i=1}^n p(y_i \mid \mathbf{w}, \mathbf{x}_i)$ and so the negative log-likelihood (that is, make this into a function of $\mathbf{w}$), which we will use as our objective function, is

$$\ell(\mathbf{w}) \overset{\text{def}}{=} -\log(p(\mathbf{y} \mid \mathbf{w}, X)) = \sum_{i=1}^n \log(1 + e^{-y_i \mathbf{w}^T \mathbf{x}_i}).$$

This is one of the most important examples of a **generalized linear model** (GLM), and our estimator $\mathbf{w}$ will be the **maximum likelihood estimator** (that is, the $\mathbf{w}$ that maximizes the likelihood, or equivalently, minimizes the negative log likelihood ℓ).

⚠ **Remark:** The above definition uses the natural log, and assumes $y \in \{-1, 1\}$. If you use $y \in \{0, 1\}$ then we need to change ℓ and $\nabla \ell$ slightly. See Appendix 2 of this homework for how to adjust your code that way.

Note: there are some numerical issues with functions like $f(a) = \log(1 + e^a)$. You don't need to worry about these for this homework, but see Appendix 3 of this homework for more details.

Problem 1: Derive

$$\nabla \ell(\mathbf{w}) = -\sum_{i=1}^{n} \sigma(-y_i \mathbf{w}^T \mathbf{x}_i) y_i \mathbf{x}_i \tag{0.1}$$

using calculus. *Hint*: first derive $\sigma'(a) = \sigma(a)(1 - \sigma(a))$.

Problem 2:

a) Derive $\nabla^2 \ell(\mathbf{w}) = \sum_{i=1}^{n} \mu_i (1 - \mu_i) \mathbf{x}_i \mathbf{x}_i^T = X^T S X$ where $S \overset{\text{def}}{=} \operatorname{diag}(\mu_i(1 - \mu_i))$.

b) Assuming all data is real-valued (no $\pm\infty$), then $0 < \mu_i < 1$. Is $\ell(\mathbf{w})$ convex? strictly convex? strongly convex? Is $\nabla \ell(\mathbf{w})$ Lipschitz continuous? If so, what constant? Prove/disprove your answers (you may assume that X is full-rank and $n \times p$ with $n \geq p$). *Hint*: the Lipschitz constant can be found as $L = \sup_{\mathbf{w}} \|\nabla^2 \ell(\mathbf{w})\|$, where $\|\cdot\|$ is the spectral norm. See wikipedia's matrix norm page for background on the spectral norm. You may want to use that it is sub-multiplicative, i.e., $\|AB\| \leq \|A\| \cdot \|B\|$.

Problem 3: Problem 4.11 (a), (b), (d) from [BV2004].

Problem 4: Brainstorm 3 possible project ideas (a title for each one), and write a few sentences with more detail on at least one of these ideas. These ideas are not binding. Project guidelines are at github.com/stephenbeckr/convex-optimization-class/blob/master/Homeworks/ProjectInformation.md.

Problem 5: List at least 2 potential partners (and their emails) for the project, and **at least one of these potential partners should be someone you did not know from before the class**. Make sure to get the partners' permission! Note: this is NOT binding, and you are not required to be partners with these people for the final project.

Problem 6: Read/skim (but do not solve) the problems from chapter 4 of [BV2004]. In particular, 4.12, 4.15, 4.23, 4.24, 4.26, 4.28, 4.40, 4.44/2.37, 4.45, 4.57, 4.59.

Homework 6: logistic regression and gradient descent

Problem 1: Make a gradient check. We will want to use $\ell(\mathbf{w})$ and $\nabla \ell(\mathbf{w})$ in our code, but first we want to be sure we implemented them correctly. Write a function `gradientCheck` which takes as input a function f, its purported gradient ∇f (also a function), and a starting point x_0 (or information about the size of its domain), and then run several checks that this indeed is the gradient.

a) Implement at least two of the tests described in **appendix 1**. Note: all of these tests use a length-scale h, and you should **make a loop where h decreases every time**, as we are interested in the behavior as $h \to 0$ [1]. You could then return your results in the form of a table, or better, plot them, similar to Fig. 2.

b) Run this gradient check code on some function/gradient pairs that you are sure you have done correctly (e.g., try a 1D linear, quadratic, and cubic — do you get the results you expect? Try a large dimensional quadratic. Is your code robust to scaling f and g to be large or small?). Run your check when you purposefully calculate the gradient slighly incorrectly. Can you use your check to reliably distinguish correct gradients?

Then run this on your logistic function/gradient pair to confirm that you implemented the gradient correctly; for the $\{\mathbf{x}_i, y_i\}$ data, either use random data but make sure both $n, p > 1$, or use the spam data from Problem 4. For example, the output might look

[1] Assuming f is well-scaled, we should not expect reliable results below about $h = 10^{-8}$ if working in double-precision; see Nocedal and Wright's chapter 8, as well as the Fig. 2.

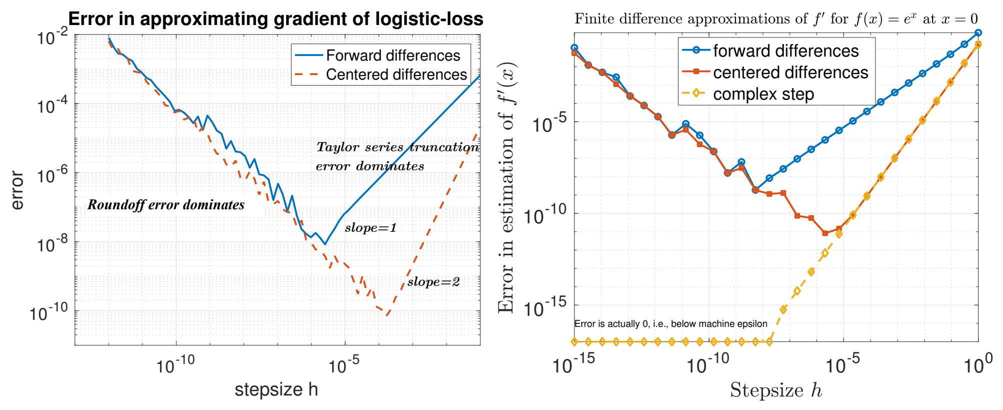

Figure 2: (Left plot) Roundoff error dominates as $h \to 0$ when using finite differences, though (if applicable) the complex-step derivative *is* numerically stable (right plot).

something like this (the rows correspond to different h, as listed in the first column, and then the columns are tests (1)–(4) including the $\mathcal{O}(h)$ sanity check from (3)):.

```
--------------------------------------------------------------------------------
h       Frwrd diff   Central diff  1st ord Taylor   2nd ord Taylor  3rd ord Taylor
--------------------------------------------------------------------------------
1.5e+00     2.6e-01     6.8e-02     1.7e+04         2.3e+04         1.6e+04
1.5e-01     2.5e-02     8.6e-04     1.3e+03         7.8e+02         8.0e+01
1.5e-02     2.5e-03     8.6e-06     2.9e+02         6.0e+00         1.5e-01
1.5e-03     2.5e-04     8.6e-08     2.5e+01         3.9e-02         1.0e-04
1.5e-04     2.5e-05     8.6e-10     2.2e+00         4.0e-04         1.0e-07
1.5e-05     2.5e-06     1.1e-10     2.7e-01         6.3e-06         1.9e-10
1.5e-06     2.5e-07     3.9e-10     2.0e-02         3.6e-08         6.9e-12
1.5e-07     2.3e-08     1.2e-08     2.7e-03         4.8e-10         7.3e-12
```

c) Note that it is hard to observe the higher-order behavior, as numerical issues will be an issue when h is small, and this break-down point is higher for higher-order methods. In the above example, the third-order method gains 3 decimal places of accuracy for every factor of 10 decrease in h, but this only happens for h in the regime 10^0 to 10^{-4}, as smaller h leads to numerical issues. For central differences, the error actually increases when $h < 10^{-5}$. See Fig. 2.

Deliverables: Please include a write-up/print-out from the results of part (b), i.e., a table or figure.

Problem 2: Code a gradient descent solver. It should take the function and gradient as input (it is up to you for the convention, but you may want to follow the convention of your favorite first-order solver from HW 4), and you should be able to control the stopping tolerance (for now, you can stop when the change in the variable from one iterate to another is small than a tolerance), the maximum number of iterations, etc.

Deliverables: Print out the most relevant part of your code, and convince me your solver works by applying it to a quadratic function $f(x) = \frac{1}{2}\|Ax - b\|^2$ with an over-determined matrix A (i.e., more rows than columns, and make it have full column rank) so that you can compare with the closed form solution $x^\star = (A^T A)^{-1} A^T b$ [to numerically compute this $x^\star$, use the backslash operator in Matlab, or `numpy.linalg.lstsq` in Python]. Show a plot of the error of your solution compared to $x^\star$ (as a function of iteration count). Label your plot well, and decide how to scale the axes (linear or logarithmic). For the stepsize of gradient descent using this function f, use a stepsize of $t = 1/\|A\|^2$ where $\|\cdot\|$ is the spectral norm.

Problem 3: Linesearch: in practice, using a constant stepsize t is restrictive and leads to sub-par performance. Using a backtracking linesearch that *never increases* the stepsize at future iterates is a quick theoretical fix when you do not know the Lipschitz constant, but also sub-par in practice.

For nonlinear programming (e.g., non-convex objectives), there are quite a few considerations, including this discouraging wisdom:

> "A line search algorithm that incorporates all these features is difficult to code. We advocate the use of one of the several good software implementations available in the public domain. See Dennis and Schnabel '83 [92], Lemaréchal '81 [189], Fletcher '87 [101], Moré and Thuente '94 [216] (in particular), and Hager and Zhang '05 [161]."
> — Nocedal and Wright, *Numerical Optimization* 2006

Standard line search **criteria** are the Wolfe conditions (consisting of the Armijo condition for sufficient decrease (see Fig 3), and a curvature condition to prevent short-steps), and the Goldstein conditions. We also need a **linesearch procedure** which tells us how to generate potential step-sizes, that are then tested against the criteria.

However, if the function is convex, and we use a backtracking linesearch procedure, then everything simplifies a bit, and we really only need to test the Armijo condition. Specifically, assume we are at the point $\mathbf{x}_k$ and have a search direction $\mathbf{p}_k$ (for gradient descent, $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$). The procedure looks like

```
1: Initial estimate $t$
2: Parameters $0 < \rho < 1$ and $0 < c < 1$          ▷ Typical values are $c = 10^{-4}$ and $\rho = 0.9$
3: while $f(\mathbf{x}_k + t\mathbf{p}_k) > f(\mathbf{x}_k) + ct\langle \nabla f(\mathbf{x}_k), \mathbf{p}_k \rangle$ do
4:     $t \leftarrow \rho t$
5: end while
```

The test statement in the "while" loop is the Armijo condition (or more exactly, the Armijo condition is when $f(\mathbf{x}_k + t\mathbf{p}_k) \le f(\mathbf{x}_k) + ct\langle \nabla f(\mathbf{x}_k), \mathbf{p}_k \rangle$). If $\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$, then the test is accepted when $f(\mathbf{x}_k + t\mathbf{p}_k) \le f(\mathbf{x}_k) - ct\|\nabla f(\mathbf{x}_k)\|^2$. In general, the second term is negative as long as $\mathbf{p}_k$ is a descent direction (specifically, $\langle \nabla f(\mathbf{x}_k), \mathbf{p}_k \rangle < 0$ is the definition of a descent direction). One can prove this linesearch procedure will terminate if ∇f is Lipschitz; for example, if $c = \frac{1}{2}$, then $t = \frac{1}{L}$ should be accepted.

What is a good estimate for the initial stepsize to try? A simple idea is to use the final stepsize from the previous step, but this can be unnecessarily small. You may want to do this, but increase the stepsize by a factor of 2 (or increase it if the previous linesearch procedure had satisfied the Armijo condition on the first try).

Deliverables: Re-run your gradient descent code on the quadratic that you used in Problem 3, but this time use a linesearch instead of a fixed stepsize. Give evidence that you found (approximately) the same solution as before.

Problem 4: Logistic regression for spam email classification (adapted from exercise 8.1 in Murphy).

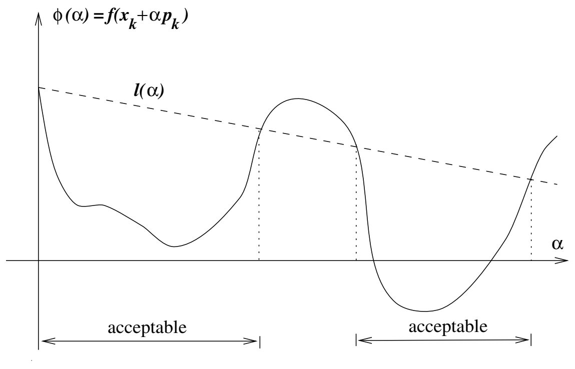

Figure 3: The sufficient decrease (Armijo) condition; adapted from Nocedal and Wright's textbook.

Consider the email spam data set discussed on p. 300 of (Hastie, Tibshirani, Friedman, *The Elements of Statistical learning* 2nd ed, 2009). This consists of 4601 email messages, from which 57 features have been extracted. These are as follows:

- 48 features, in $[0, 100]$, giving the percentage of words in a given message which match a given word on the list. The list contains words such as "business", "free", "george", etc. (The data was collected by George Forman of HP labs, so his name occurs quite a lot.)
- 6 features, in $[0, 100]$, giving the percentage of characters in the email that match a given character on the list. The characters are `; ( [ ! $ #`
- Feature 55: The average length of an uninterrupted sequence of capital letters (max is 40.3, mean is 4.9)
- Feature 56: The length of the longest uninterrupted sequence of capital letters (max is 45.0, mean is 52.6)
- Feature 57: The sum of the lengths of uninterrupted sequence of capital letters (max is 25.6, mean is 282.2)

Load the data from `spamData.mat`, which contains a training set (of size 3065) and a test set (of size 1536). You can get this online in Matlab format at `https://github.com/probml/pmtk3/tree/master/data/spamData` or in plain text at ftp.ics.uci.edu. If you use the plain text version, there is no predefined testing/training set, so just shuffle the data then pick a training set of size 3065 and use the rest for testing (or, use the Matlab format data along with `scipy.io.loadmat`).

There are different methods to pre-process the data, e.g., standardize the columns of X (we can standardize the rows of X, but this just is absorbed into $\mathbf{w}$). For this problem, transform the features using $\log(x_{ij} + 0.1)$. One could also add some regularization to the loss function (e.g., add $\lambda/2\|\mathbf{w}\|^2$ for a small λ) which can help generalization error, but this is not necessary. Also note that you need to transform the labels from $\{1, 0\}$ to $\{1, -1\}$.

a) Implement $\nabla\ell(\mathbf{w}) = -\sum_{i=1}^{n} \sigma(-y_i\mathbf{w}^T\mathbf{x}_i)y_i\mathbf{x}_i$ using this spam data for the X and $\mathbf{y}$. Do not write a `for` loop and implement (0.1) directly, rather rewrite this to use matrix multiplication instead of a sum, as the code will be much faster. One way to do this is to define $\boldsymbol{\mu} = (\mu_i)_{i=1}^n$ with $\mu_i = \sigma(y_i\mathbf{w}^T\mathbf{x}_i)$ and then $\nabla\ell(\mathbf{w}) = -X^T(\mathbf{y}\odot(1-\boldsymbol{\mu}))$ where $\odot$ represents element-wise multiplication (`.*` in Matlab). For the spam data set, you should be able to evaluate a gradient in a second or less (my implementation in Matlab takes 0.0065 seconds on my laptop). **Deliverables:** your code, and the time (in seconds) of how long it takes to evaluate at an arbitrary point $\mathbf{w}$.

b) Run your gradient descent solver with the logistic function (using the training data only) to obtain a classifier $\mathbf{w}$. **Deliverables:** relevant parts of your code, and a plot of function value vs iteration (you may use fminunc_wrapper_simple.m to help with this).

c) Using the (nearly) optimal $\mathbf{w}$ you found, make your classification according to whether $\sigma(\mathbf{w}^T\mathbf{x}_i) > 0.5$ or not. Your misclassification rate should be around 0.052 on the training data, and around 0.059 on the testing data, which is not bad! **Deliverables:** a paragraph that discusses your final results (see above paragraph) and your misclassification rate, and any issues or comments. **Optional**: which features are most important? (the labels are at ftp.ics.uci.edu).

Optional Re-run this using a fancier algorithm (which you do not need to code yourself), e.g., in Matlab, Schmidt's `minFunc` uses L-BFGS by default, or use Goldstein's `FASTA` which uses a Nesterov accelerated method. This is also a good candidate for Newton's method (it's not too high dimensional; you'd want to use an existing implementation of Newton's method that has a linesearch in it, since a pure Newton method is not a good idea).

Appendix 1: Gradient checks

Note that we are going back to optimization notation (where $\mathbf{x}$ is the variable), so forget the machine learning notation for now (where $\mathbf{w}$ was the variable and $\mathbf{x}_i, y_i$ were data).

1. **Forward finite difference approximation**. Approximate the gradient vector g using $g_i \approx \frac{f(\mathbf{x}_0+h\mathbf{e}_i)-f(\mathbf{x}_0)}{h}$ where $\mathbf{e}_i$ is the i^{th} unit vector, and then compare this to $\nabla f(\mathbf{x}_0)$. The error should decay like $\mathcal{O}(h)$

when h is not too small (but recall from your undergrad numerics class: as $h \to 0$, roundoff error starts to dominate, so the approximation actually gets worse as h gets smaller!)

2. **Centered finite difference approximation**. Same as above, but use a centered-difference approximation $g_i \approx \frac{f(\mathbf{x}_0+h\mathbf{e}_i)-f(\mathbf{x}_0-h\mathbf{e}_i)}{2h}$; now, the error should decay like $\mathcal{O}(h^2)$ for a while (again, at some point roundoff error will start to dominate).

3. This check is faster as we only evaluate f at one new point. First, we could note a sanity check (but this doesn't actually test the implementation of our gradient): pick $\mathbf{x}_1$ in the domain of f, and then you could verify that $|f(\mathbf{x}_0) - f(\mathbf{x}_0 + h\mathbf{x}_1)| = \mathcal{O}(h)$. Secondly, the real check is that we note $|f(\mathbf{x}_0) + \langle \nabla f(\mathbf{x}_0), h\mathbf{x}_1\rangle - f(\mathbf{x}_0 + h\mathbf{x}_1)| = \mathcal{O}(h^2)$ (assuming the Hessian is smooth). See Dolfin Adjoint documentation for details. This method does not suffer numerical instability the way that the forward and centered finite difference approximations do.

4. Combining the two parts of 3 above, and assuming f has 3 continuous derivatives, then we can make an $\mathcal{O}(h^3)$ check as follows. Pick $\mathbf{x}_1$ as in 3, and let $\widetilde{\mathbf{x}} = \mathbf{x}_0 + h\mathbf{x}_1$, then:

$$\begin{aligned} f(\widetilde{\mathbf{x}}) &= f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\widetilde{\mathbf{x}} - \mathbf{x}_0) + \frac{1}{2}(\widetilde{\mathbf{x}} - \mathbf{x}_0)^T \nabla^2 f(\mathbf{x}_0)(\widetilde{\mathbf{x}} - \mathbf{x}_0) + \mathcal{O}\left(\|\widetilde{\mathbf{x}} - \mathbf{x}_0\|^3\right) \\ f(\mathbf{x}_0) &= f(\widetilde{\mathbf{x}}) + \nabla f(\widetilde{\mathbf{x}})^T(\mathbf{x}_0 - \widetilde{\mathbf{x}}) + \frac{1}{2}(\mathbf{x}_0 - \widetilde{\mathbf{x}})^T \nabla^2 f(\mathbf{x}_0)(\mathbf{x}_0 - \widetilde{\mathbf{x}}) + \mathcal{O}\left(\|\mathbf{x}_0 - \widetilde{\mathbf{x}}\|^3\right). \end{aligned}$$

Subtracting these equations gives

$$\begin{aligned} -2\left(f(\widetilde{\mathbf{x}}) - f(\mathbf{x}_0)\right) + \left(\nabla f(\mathbf{x}_0) + \nabla f(\widetilde{\mathbf{x}})\right)^T \left(\widetilde{\mathbf{x}} - \mathbf{x}_0\right) &= \mathcal{O}\left(\|\widetilde{\mathbf{x}} - \mathbf{x}_0\|^3\right) \\ &= \mathcal{O}(h^3). \end{aligned}$$

5. **Complex-step finite difference approximation**. The complex-step derivative[2] is like a centered difference approximation but doesn't have the numerical roundoff issues; it is applicable only if your code can handle complex numbers. This is based on the observation that (for now, let f be a 1D function)

$$\begin{aligned} f(x+ih) &= f(x) + ihf'(x) + \frac{(ih)^2}{2!}f''(x) + \frac{(ih)^3}{3!}f'''(\xi) \\ &= f(x) - h^2/2f''(x) + hi\left(f'(x) - h^2/6f'''(\xi)\right) \end{aligned}$$

for some ξ between x and $x+ih$, thus assuming the third derivative is bounded, $f'(x) = \operatorname{Im}\left(h^{-1}f(x+ih)\right) + \mathcal{O}(h^2)$. See below for details (e.g., we require f is analytic). In particular, we assume $f(x) \in \mathbb{R}$ if $x \in \mathbb{R}$, but we also need $f(x+ih) \in \mathbb{C} \setminus \mathbb{R}$ if $x, h \in \mathbb{R}$ otherwise the test clearly makes no sense (since the imaginary part of would be zero).

See Fig. 2 (right), motivated by these slides by Peder Olsen.

However, be careful: the complex step derivative **only works if f is analytic**. See "The Complex-Step Derivative Approximation" by Martins, Sturdza and Alonso '03 for a derivation. Most building blocks of functions, like trig, exponentials, $+, \times$, etc. are analytic, with the notable exception of the absolute value function: hence any ℓ_p norm cannot be analytic. In particular, if $f(x+iy)$ is real-valued (even with complex input), then if f were complex analytic, by the Cauchy-Riemann equations, f would have to be the constant function on the real line. So *the complex step derivative should not be used for least-squares based objective functions*; however, if the objective is $\|\mathbf{x}\|_2^2$ (and not $\|\mathbf{x}\|_2$) then you can build an alternative implementation via $\sum_i x_i^2$ (rather than $(\sqrt{\sum_i |x_i|^2})^2$) which agrees for real-valued input but is analytic since it is a polynomial.

[2]there is a lot of material online about this; for example, Tim Vieira's blog post

Appendix 2: Logistic Regression for Classification, $y_i \in \{0,1\}$

For your reference, we derive the logistic loss when the labels y_i are in $\{0,1\}$ and not $\{-1,1\}$.

In the case $y_i \in \{0,1\}$, logistic regression corresponds to the following binary classification model:

$$p(y|\mathbf{x},\mathbf{w}) = \mathrm{Ber}(y|\sigma(\mathbf{w}^T\mathbf{x})) = (\sigma(\mathbf{w}^T\mathbf{x}))^{\mathbb{I}(y_i=1)}(1-\sigma(\mathbf{w}^T\mathbf{x}))^{\mathbb{I}(y_i=0)} \tag{0.1}$$

where $\sigma(\mathbf{w}^T\mathbf{x}) = \dfrac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})} = (1+\exp(-\mathbf{w}^T\mathbf{x}))^{-1}$. For simplicity define $\mu_i = \sigma(\mathbf{w}^T\mathbf{x}_i)$ (note that this is not quite the same μ_i that we used for $y_i \in \{-1,1\}$). The negative-log likelihood is then given by:

$$\begin{aligned}
\ell(\mathbf{w}) &= -\sum_{i=1}^{N} \log[\mu_i^{\mathbb{I}(y_i=1)} \cdot (1-\mu_i)^{\mathbb{I}(y_i=0)}] \\
&= -\sum_{i=1}^{N} [y_i \log(\mu_i) + (1-y_i)\log(1-\mu_i)] \\
&= -\sum_{i=1}^{N} [y_i \log((1+\exp(-\mathbf{w}^T\mathbf{x}_i))^{-1}) + (1-y_i)\log(\exp(-\mathbf{w}^T\mathbf{x}_i)\cdot(1+\exp(-\mathbf{w}^T\mathbf{x}_i))^{-1}).
\end{aligned}$$

Using properties of the logarithm and some algebra, this simplifies to:

$$\begin{aligned}
\ell(\mathbf{w}) &= -\sum_{i=1}^{N} -y_i \log(1+\exp(-\mathbf{w}^T\mathbf{x}_i)) + (1-y_i)(-\mathbf{w}^T\mathbf{x}_i) - (1-y_i)\log(1+\exp(-\mathbf{w}^T\mathbf{x}_i)) \\
&= \sum_{i=1}^{N} \log(1+\exp(-\mathbf{w}^T\mathbf{x}_i)) + (1-y_i)\mathbf{w}^T\mathbf{x}_i.
\end{aligned}$$

Now, we derive the derivative with respect to our decision variable $\mathbf{w}$. For the j-th component, we have:

$$\begin{aligned}
\frac{\partial \ell(\mathbf{w})}{\partial w_j} &= \sum_{i=1}^{N} \frac{-\exp(-\mathbf{w}^T\mathbf{x}_i)x_{ij}}{1+\exp(-\mathbf{w}^T\mathbf{x}_i)} + (1-y_i)x_{ij} \\
&= \sum_{i=1}^{N} \sigma(\mathbf{w}^T\mathbf{x}_i)x_{ij} - y_i x_{ij}.
\end{aligned}$$

That is,

$$\nabla \ell(\mathbf{w}) = \sum_i (\mu_i - y_i)\mathbf{x}_i = \mathbf{X}^T(\mu - \mathbf{y}). \tag{0.2}$$

The Hessian can be found in a similar manner.

Appendix 3: Numerical Stability of some log & exp functions

With the function $f(a) = \log(1+e^a)$, we have some numerical issues that we have to be aware of whenever $|a|$ is large. Suppose $a = 400 \cdot \log(10)$, then $e^a = 10^{400}$ is huge, and $e^a \approx 1 + e^a$ so we expect $f(a) \approx a$. However, the intermediate calculation of e^a gives 10^{400}, which is numerical *overflow* (i.e., in double precision, this number is too large to store. In Matlab, you can see the limit with `realmax`, and is about 1.7977e+308). So this number is treated as Infinity or NaN, and we cannot take its log. To get around this, when a is large, can write $\log(1+e^a) = \log(e^a(e^{-a}+1)) = a + \log(e^{-a}+1)$. This leads to our second problem: how to find $f(a)$ when a is small (i.e., very negative). If $a = -20\log(10)$ then $e^a = 10^{-20}$, and numerically, $1 + 10^{-20} = 1$ because 10^{-20} is smaller than the "machine epsilon" (in Matlab, with double precision, this is `eps`, and about 2.2204e-16), so evaluating $f(a)$ would just give $\log(1) = 0$. To fix this, use Matlab's `log1p` function (or `numpy.log1p` in Python) which calculates $\log(1+\epsilon)$ accurately for small ϵ using a smarter method such as an explicit Taylor series expansion.

By the way, a similar numerical issue arises when we want to find $f(a,b) = \log(e^a + e^b)$, or more generally, $f(x) = \log(\sum_i e^{a_i})$, known as the log-sum-exp function for obvious reasons. If $a \gg b$, then $e^a \gg e^b$ and we lose precision when we add these, so $f(a,b) \approx \log(e^a)$. We can restore some precision by multiplying by e^{-a}, much as we did in the above paragraph. In general, we subtract off $\max_i x_i$; this is known as the **"log-sum-exp trick."** In Python, this is implemented for you by `numpy.logaddexp`. Many of these log-add-exp/log-sum-exp functions are implemented so that evaluating them on the $n \times 2$ matrix $[a,b]$ gives $\sum_{i=1}^{n} \log(e^{a_i} + e^{b_i})$, which can be converted to $\sum_{i=1}^{n} \log(1 + e^{a_i})$ by using $b \equiv 0$.

You don't need to worry about these numerical issues for this homework (unless you want to go above-and-beyond), I'm just mentioning them to make you aware of them.