Assignment 5, CSCI 471/571 Fall 2020

Your name here

Due: Wednesday November 18, 11:59 p.m.

Group work following the guidelines in the Syllabus is okay.
Github username: user_name
List your collaborators.

MNIST problem adapted from Kevin Jamieson and Jamie Morgenstern's course at UW.

Total points: 52

The following section should be computed by hand. Show your work.

Logistic loss

1. *[10 total points]* Let $\phi(z) = \frac{1}{1+e^{-z}}$ be the logistic function. In class, we talked about how logistic regression models the probability that an example $\vec{x}$ has label $Y = +1$ as $P[Y = +1|\vec{x}, \vec{w}] = \phi(b + \vec{x}^T\vec{w})$, i.e. by passing the linear prediction through the logistic function.

The *logistic loss function* for a linear classifier is defined as

$$L_{\text{logistic}} = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y_i(b + \vec{x}_i^T\vec{w}))).$$

The goal of this problem is to show that the gradients of the logistic loss are

$$\nabla_{\vec{w}} L_{\text{logistic}} = -\frac{1}{n}\sum_{i=1}^{n} P[Y_i = -y_i|\vec{x}_i, \vec{w}]y_i\vec{x}_i \tag{1}$$

and

$$\nabla_b L = \frac{\partial L_{\text{logistic}}}{\partial b} = -\frac{1}{n}\sum_{i=1}^{n} P[Y_i = -y_i|\vec{x}_i, \vec{w}]y_i. \tag{2}$$

Hints: (a recommended plan of attack) It is sufficient to get the gradient when there is $n = 1$ data point $(\vec{x}, y)$ and then take the average. You can consider the derivative with respect to b as a special case of the gradient when you think of augmenting $\vec{x}$ with an entry equal to 1.

1. *[2 points]* Show that $P[Y = -1|\vec{x}, \vec{w}] = \phi(-(b + \vec{x}^T\vec{w}))$.

2. *[2 points]* Write "the probability that random variable Y_i is different than the y_i that you observe" i.e. $P[Y_i = -y_i|\vec{x}_i, \vec{w}]$ in terms of the margin $z_i = y_i(b + \vec{x}_i^T\vec{w})$. Your answer should be using the function ϕ, without containing any exponentials.

3. *[4 points]* Derive the gradients given in (1) and (2).

4. *[2 points]* Combine your results from 1.2 and 1.3 to rewrite (1) and (2) in terms of the margins $z_i = y_i(b + \vec{x}_i^T\vec{w})$. Briefly explain why this is a practical way to evaluate the gradient.

Logistic regression for MNIST

For the following, you should program in Python and may use the `numpy` package and `matplotlib` for plotting but *you may not use any of the built-in least-squares solvers or anything from* `sklearn`. Any output from your program (displays of matrices or vectors, error rates, plots) must be included in your pdf submission. Your code must be submitted as described in the Syllabus. All plots should be legible with axes labeled and legends if there are multiple things plotted.

Get the MNIST data from `http://yann.lecun.com/exdb/mnist/` and install the package `https://pypi.python.org/pypi/python-mnist`. An easy way to install this package is by `pip install python-mnist`. The starter code shows how to load the dataset.

MNIST is a dataset containing training and testing samples of handwritten digits 0–9 that are cropped and centered 28×28 pixel images ($d = 784$). It is a canonical dataset for classification. We will focus on just binary classification, where we would only like to predict whether a digit is a 2 or a 7. Remove all other examples from the training and testing datasets. Let $Y = +1$ for all the 7's in the dataset and $Y = -1$ for the 2's. There are some hints in the starter code for doing this kind of subselection.

We will apply ℓ_2-regularized logistic regression to this problem. Given a binary classification dataset $\{(\vec{x}_i, y_i)\}_{i=1}^n$ with $\vec{x}_i \in \mathbb{R}^d$ and $y_i \in \pm 1$, the regularized cost function is

$$C(\vec{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y_i(b + \vec{x}_i^T\vec{w}))) + \lambda\|\vec{w}\|_2^2. \qquad (3)$$

For all experiments set $\lambda = 0.1$. Don't miss the $\frac{1}{n}$ in front of the loss.

2. *[26 total points]* We apply logistic regression to the MNIST problem with a linear predictor.

   1. *[2 points]* Derive the gradients $\nabla_{\vec{w}} C$ and $\frac{\partial C}{\partial b}$ in a similar form as 1.4. *This should be easy given what you know from problem 1 and past homeworks.*

   2. *[10 points]* Implement gradient descent with an initial iterate of all zeros for $\vec{w}$ and b. Experiment with step sizes until you find one that converges on the training set as fast as possible (something between 0.001 and 0.5 should work). Since you will be evaluating the gradients many times, ensure that your numpy code to evaluate the gradient is vectorized. Report the following, along with the step size:

      - Figure 1: Plot the cost (loss + regularization) evaluated on both the training and the test set as a function of iteration number. Show both curves on the same plot. Remember, you are only optimizing the training cost.
      - Figure 2: For the training and the test set, classify the points using the rule $\text{sign}(b + \vec{x}_i^T\vec{w})$ and plot the error rate (the fraction of predictions which were incorrect, scaled to be a percent is useful) also as a function of the iteration number.

   3. *[7 points]* Repeat 2.2 using stochastic gradient descent with a batch size of 1. Be careful that the expected gradient (with respect to random selection of data) should be equal to the gradient in 2.1. Take careful note of how to scale the regularizer.

4. *[7 points]* Repeat 2.2 using stochastic gradient descent with a batch size of 100. Instead of approximating the gradient with a single random example, use 100 random examples (see `np.random.choice`).

3. *[16 total points]* Now you will investigate how much better you can do on this MNIST task using a simple kind of nonlinear featurization. Neural networks where the first layer weights are untrained end up being equivalent to linear models, just like when we formed polynomial features and fit polynomial models using ridge regression. Here we will use a cosine nonlinearity, which is different than the common ones used in networks but has connections to kernel methods.

We will think of taking the original d-dimensional features (pixel intensities) and *nonlinearly transforming* them into a p-dimensional second layer, where the p-dimensional vectors representing 2's and 7's are more easily separable. Use:

$$\vec{h} = \cos(G\vec{x} + \vec{c}) \in \mathbb{R}^p\,, \tag{4}$$

where $\vec{c} \in \mathbb{R}^p$ is a vector with entries c_i sampled iid uniformly from $[0, 2\pi]$ (use `np.random.rand` and rescale), $G \in \mathbb{R}^{p\times d}$ is a vector with entries G_{ij} sampled iid from $\mathcal{N}(0, \sigma^2)$ (use `np.random.randn` and rescale; note that σ^2 is the variance of the random variable and that you should rescale by the standard deviation), and the cosine nonlinearity is applied pointwise.

We will assume a logistic regression readout for predicting the labels. In other words, once you form the random features (4), you will predict the output as $P[Y = +1|\vec{x}] = \phi(\vec{h}^T\vec{w})$ using the random features $\vec{h}$ instead of the original ones $\vec{x}$. Fitting this model is easy: Just form an $n \times p$ matrix H from your original $n \times d$ data X, and feed it into the gradient descent function you wrote in 2. This is a one-liner in numpy: `H_train = np.cos(X_train @ G.T + c)`. Note that to make a prediction at a point $\vec{x}$, you will also have to transform it in this way.

Your goals here are to use cross-validation to select the parameter σ and estimate the error rate on held-out data. The hyperparameter σ determines the variance in the frequencies of the cosines, and you can think of it as a hyperparameter related to the bandwidth in the kernels that we have seen. For your experiments, set $p = 3000$ and again $\lambda = 0.1$.

1. *[10 points]* We will perform model selection using only the training data. Reserve 80% of the training set for fitting and 20% for validation. For a range of σ spanning 10^{-3} to 10^2 by single orders of magnitude (see `np.logspace`), fit separate logistic regression models using random features with these weights to just this training set. Experiment with the step size and any convergence conditions (i.e. $\|\vec{w} - \vec{w}_{\text{old}}\|_\infty$ or max number of iterations) for a fixed variance (e.g. $\sigma = 1$) to find a value that works before running the sweep. Plot the error rates on the training and validation sets as a function of σ.
2. *[2 points]* Select the σ that you think will generalize best and briefly explain why you picked it.
3. *[2 points]* Estimate the error rate on new data. Using the selected σ, refit to *all* of the training data (the 80% + 20% for validation, this is okay because we aren't using this data to estimate the error rate, just select and fit the model). Evaluate the performance on the test set and report the error rate here.
4. *[2 points]* Comment on the differences in performance compared to the linear model of problem 2. Talk about both the ability of the model to fit the data (bias) and its ability to generalize (variance).

Note that p and λ are also hyperparameters and one could conceivably perform model selection over them as well. But what are the downsides of having to optimize over more hyperparameters? If your results are noisy, you could also average over multiple networks, or use other cross-validation methods. You may need to use a different step size on this problem than the other one, but the optimal step size should not vary too much with σ.