Handin 1, Machine Learning 2015

September 2, 2015

Intro

In this handin you will build classifiers for optical character recognition (OCR). You should use Python and NumPy for the programming parts. The models we use can be implemented with relatively few lines of actual code if expressed using matrices and vectors (Python and NumPy are **much much** faster when you express your computation using matrix products instead of for-loops), so the key is in understanding and matrix products. In this document we go through the models in quite a lot of detail to ease the understanding. We use matrix and vector notation throughout since it simplifies the exposition, so you might as well get used to it.

You can write your final report in Danish if you prefer. The maximal report length is 5 pages. You are allowed to be up to 3 members in a group. You are encouraged to discuss the exercise between groups and help each other as much as possible without of course copying each other's work. Particularly, discussing the quality of your classifiers is probably a good idea to get an indication if you are doing it correctly. For these discussions and additional questions use the discussion forum on the BlackBoard course site. Bonus questions may be skipped but you are encouraged to try and at least think about answering them.

There are associated data files and examples for you on the website.

mnistTrain.npz A training set of MNIST digits.

mnistTest.npz A test set of MNIST digits.

auTrain.npz A training set with the images you produced in class. It comes online when we have processed your inputs.

When you hand in your report you must also upload code. We will specify what we need after the description of the exercise.

Be sure to discuss the choices you have made in your implementation in your report.

Logistic Regression

In logistic regression we model the target function as a probability distribution $p(y \mid x)$. This probability distribution is defined by the logistic function $\sigma(z) = 1/(1+e^{-z})$ over a linear function of the input data, $z = \sum_{i=0}^{d} \theta_i x_i = \theta^\intercal x$, that is parameterized by the vector θ. As in the lectures and the first coding exercise, we encode the bias variable into the input vectors as x_0 and force $x_0 = 1$ on all data points. The logistic function is a nice smooth function with simple derivaties, $\frac{\partial \sigma}{\partial z} = (1 - \sigma(z))\sigma(z)$, which makes it pleasing to work with. The model becomes,

$$p(y \mid x, \theta) = \begin{cases} \sigma(\theta^\intercal x) & \text{if } y = 1 \\ 1 - \sigma(\theta^\intercal x) & \text{if } y = 0, \end{cases}$$

that is, the probability that $y = 1$ given x and θ is the probability of getting heads with a biased coin, where the bias is $\sigma(\theta^\intercal x)$. Given a fixed θ we can use the function p to make classification by returning the most likely class, that is, given x:

$$\text{return 1 if } \sigma(\theta^\intercal x) \geqslant 0.5, \text{otherwise return 0,}$$

Since $\sigma(z) = 0.5$ when $z = 0$, we end up with a simple linear classifier e.g.

$$\text{return } \theta^\intercal x \geqslant 0$$

The job at hand is to find a good θ and we will do that using the *Maximum Likelihood method*.

The input is a labeled dataset $D = (X, Y) = \{(x_i, y_i) \mid i = 1, \ldots, n\}$, where $x_i \in \{1\} \times \mathbb{R}^d$ is a column vector of length $d + 1$ (a $(d + 1) \times 1$ matrix) and $y_i \in \{0, 1\}$ is a number. As stated above, we use the first input dimension for the bias 1. Think of X as a matrix where each row is an input point (column vector transposed) x_i, and y as a column vector (matrix with one column) where the i'th entry is the class of input point x_i. Notice that y_i's are not probabilities, but actual realisation of events. We assume (as always) that the points in D are independently sampled. This means that under our model, for a fixed vector of parameters θ, the likelihood of the data is

$$p(D \mid \theta) = \prod_{(x,y)\in D} p(y \mid x, \theta) = \prod_{(x,y)\in D} \sigma(\theta^\intercal x)^y (1 - \sigma(\theta^\intercal x))^{1-y}$$

Notice the last equality, which gives a convenient way of expressing the probability $p(y \mid x, \theta)$ as a product. You should convince yourself that it is true; recall that y is either 0 or 1.

Computing the Maximum Likelihood Parameters

We want to compute the parameters that makes the likelihood as large as possible, hence the name Maximum Likelihood, and it is defined as

$$\theta_{\text{ml}} = \arg\max_\theta p(D \mid \theta).$$

Your task is to find θ_{ml}, or at least parameters that are very close to that. To make this task simpler we minimize the negative log likelihood (NLL) of the data instead. This transforms the likelihood product into a pointwise sum which is a lot easier to handle. The negative log likelihood is

$$\text{NLL}(D \mid \theta) = -\sum_{(x,y)\in D} y \ln(\sigma(\theta^\intercal x)) + (1 - y) \ln(1 - \sigma(\theta^\intercal x))$$

which is called the *cross entropy* error function. This is the function you must minimize. Before we start implementing an algorithm that minimizes the negative log likelihood, we need to know what we are dealing with. Basically, if a function is convex/concave we know we can optimize it efficiently. Luckily that is actually the case, but you should not take my word for it (Bonus Question 1 below).

Now that you know the negative log likelihood function is convex you know that a local minimum is a global minimum and at a local minimum the gradient (if it exists) vanishes, that is, is zero. Luckily the negative log likelihood function is smooth and the gradient is (skipping all the derivations)

$$\nabla\text{NLL}(\theta) = \frac{\partial \text{NLL}}{\partial \theta} = \left[\frac{\partial \text{NLL}}{\partial \theta_0}, \frac{\partial \text{NLL}}{\partial \theta_1}, \ldots, \frac{\partial \text{NLL}}{\partial \theta_d}\right] = \cdots = -X^\intercal(Y - \sigma(X\theta)),$$

where σ applied to a vector means entrywise application of the sigmoid function (the logistic function). You should of course verify that this is true. Notice how we compute the gradient,

which is a vector of size $d+1$, in one matrix product. Unfortunately, we cannot analytically solve to find a zero for this gradient (as we could for linear regression). Instead we turn to gradient descent optimization algorithms.

Logistic Regression Implementation

Now you must implement logistic regression and apply it to the given OCR data. We have defined a number of subtasks that should help you, especially with debugging.

- Write code so you may load and visualize the data. Use pcolormesh or imshow commands to visualize input digits (and later weight vectors)
- Make a small data set you can use for testing implementation correctness. Testing all data may take a long time.
- Implement cost function and gradient computation. Both are functions of the data and the parameters (D, θ). You can test your gradient computation by computing the gradient numerically. That is, the gradient of function f at input x may be approximated numerically as
$$\frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon}$$
for small ε. See `https://en.wikipedia.org/wiki/Numerical_differentiation` for more details.
- Implement gradient descent using the gradient computation you just did. After each iteration print the new cost function to the screen or on a plot. It should be decreasing. Think about the line search method you use and always verify that it works by looking at the cost function as a function of the number of iterations.
- Use your logistic regression implementation to train a classifier for digit recognition (for instance 2 vs. 7) that when given x returns $\arg\max_y p(y \mid x, \theta)$. In this case we care about the percentage of correct classifications, not the value of the cost function. How does the weight vector θ look as the algorithm progresses and when the algorithm has terminated (plot as a 2D image)? Try different pairs of digits. Does the weight vector θ behave like you would expect?
- Implement a full classifier for digit recognition using the one-vs.-all technique.
- Add $L2$ ($\lambda \sum_{i=1}^{d} \theta_i^2$) regularization to the cost function and gradient. You should work out the equations yourselves Notice that we did not penalize the bias term θ_0. Implement model selection using validation (or cross-validation) to find the best hyperparameters for regularization. Use a grid search e.g. test something (not necesarily exactly) like $\lambda = 3^i, i = -6, \ldots, 5$ and include $\lambda = 0$ (e.g. unregularized might be best).

Report In your report, you should explain and discuss your final algorithm, in particular how you decide on your line search procedure. Also plot some of the digits that your classifiers make mistakes on and discuss why this may be. Your report should include a plot of the cost function as a function of the iteration, and a figure that shows the plot of the parameter vectors for the 10 all vs. one classifiers. Furthermore, your report should include a table of results for the pairwise computations as well as a table of results for the full classifier. Include the results for the regularized versions of the cost function and gradient and compare the result with the unregularized implementation.

Furthermore you should answer the following theoretical questions (although the bonus question is optional).

Sanity Check What happens if we randomly permute the pixels in each image (with the same permutation) before we train the classifier? Will we get a classifier that is better, worse, or the same? Give a short explanation.

Linear Separable If the data is linearly separable, what happens to weights when we implement logistic regression with gradient descent? Assume that we have full precision (that is, ignore floating point errors). We can run gradient descent on the data set for as long as we want. Now what will happen with the weights in the limit? Do they converge to some fixed number (fluctuate around it) or do they keep increasing in magnitude (absolute value)? Give a short explanation for your answer. What happens if we add regularization?

Bonus Question: Convexity of negative log likelihood Show that the negative log likelihood function for logistic regression is convex. Is it still convex if we add regularization?

Code Requirements In order for us to evaluate your code, you should hand-in a Python file with the following functions. Note that the file should not execute any functions when imported as a module, so you should wrap the call to your main function in `if __name__ == "__main__"`. For more information see the Python documentation on `__main__`.

Cost Function and Gradient function named `log_cost` that takes 3 inputs, data X, target y, parameters θ. X is a $n \times (d+1)$ matrix, y is an $n \times 1$ column vector and θ is a $(d+1) \times 1$ column vector. The function must return the cost function E_{in} and the gradient ∇E_{in} when applied to data X, y with parameters θ.

Gradient Descent Algorithm Function named `log_grad` that takes three arguments X, y, θ as above and runs gradient descent and return the best parameters found on the data. The algorithm should be optimized for the auDigits training set.

Best Parameters Found Besides your Python file, save the best parameters you have found in a file with `np.savez("params.npz", theta=best_theta)` and hand it in. The parameters should be optimized for the auDigits data set classifying whether a digit is 2 or 7.

Multinomial/Softmax Regression

In this exercise we generalize logistic regression to handle K classes instead of 2. This is known as Multinomial or Softmax regression and we will use the exact same approach as for logistic regression, but it becomes a little more technical due to the extra classes. In Softmax regression we are classifying into K classes (instead of 2 for logistic regression). We encode the target values, y, as a vector of length K with all zeros except one which corresponds to the class. If an example belong to class 3 and there are five classes then $y = [0, 0, 1, 0, 0]^\intercal = e_3$. So Y is now a matrix of size $n \times K$, and X is unchanged.

We can no longer use the logistic function as the probability function. Instead we use a generalization which is the *softmax* function. Softmax takes as input a vector of length K and outputs another vector of the same length K, that is a mapping from the K input numbers into K *probabilities* e.g. they sum to one. It is defined as

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \text{ for } j = 1, \ldots, K.$$

Notice that the denominator acts as a normalization term that ensures that the probabilities sum to one. Again we get nice derivatives (which we like),

$$\frac{\partial \operatorname{softmax}(x)_i}{\partial x_j} = (\delta_{i,j} - \operatorname{softmax}(x)_j) \operatorname{softmax}(x)_i,$$

where $\delta_{i,j} = 1$ if $i = j$ and 0 otherwise. As before we use a linear model for each class. The parameter set θ is represented as a $(d+1) \times K$ matrix giving $d+1$ parameters ($+1$ for the bias) for each of the K classes (parameters for class c is column c), meaning that $\theta = [\theta_1, \ldots, \theta_K]$. We get

$$p(y \mid x, \theta) = y^\intercal x = \begin{cases} \operatorname{softmax}(\theta^\intercal x)_1 & \text{if } y = e_1, \\ \vdots & \\ \operatorname{softmax}(\theta^\intercal x)_K & \text{if } y = e_K. \end{cases}$$

Think of the probability distribution over y as throwing a K-sided die where the likelihood of landing on each of the K sides is stored in the vector $\theta^\intercal x$ (which is a vector of length K) and the probability of landing on side i is $\operatorname{softmax}(\theta^\intercal x)_i$.

As for logistic regression we compute the likelihood of the data given a fixed matrix of parameters. We use the notation $[z]$ for the indicator function e.g. $[z]$ is one if z is true.

$$P(D \mid \theta) = \prod_{(x,y) \in D} \prod_{j=1}^{K} \operatorname{softmax}(\theta^\intercal x)_j^{[y_j = 1]} = \prod_{(x,y) \in D} y^\intercal \operatorname{softmax}(\theta^\intercal x).$$

This way of expressing is the same as we did for logistic regression. The product over the K classes will have one element that is not one namely the y_j'th element (y is a vector of $K-1$ zeros and a one). The remaining probabilities are raised to a power of zero and has the value one.

For convenience we minimize the negative log likelihood of the data instead of maximizing the likelihood of the data and get a pointwise sum.

$$\operatorname{NLL}(D \mid \theta) = -\sum_{(x,y) \in D} \sum_{j=1}^{K} [y_j = 1] \ln(\operatorname{softmax}(\theta^\intercal x)_j) = -\sum_{(x,y) \in D} y^\intercal \ln(\operatorname{softmax}(\theta^\intercal x)).$$

As for logistic regression this is a convex function but we cannot solve for a zero analytically and turn to gradient methods. To use gradient descent as before all you really need is the gradient of the negative log likelihood function. This gradient is a *simple* generalization of the one used in logistic regression. We now have a set of parameters for each class, θ_j for $j = 1, \ldots, K$ (the j'th column in the parameter matrix θ). Luckily some nice people tell you what it is:

$$\nabla \operatorname{NLL}(\theta) = \cdots = -X^\intercal (Y - \operatorname{softmax}(X\theta)),$$

where softmax is taken on each row of the matrix (that is, $X\theta$ is an $n \times K$ matrix and we need to compute softmax for each training case over the K classes). You should, of course, verify this yourself but you do not have to prove it. This should look awfully familiar to you from the logistic regression Exercise. With this in place you should be able to make a gradient descent implementation for softmax regression, like you did for logistic regression.

Numerical Issues with Softmax

There are some numerical issues with the softmax function,

$$\operatorname{softmax}(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \text{ for } j = 1, \ldots, K,$$

because we are computing a sum of exponentials (before taking logs again), and when we exponentiate numbers they tend to become very big, giving numerical problems. Let's look at the function for a fixed j, $\frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}}$. Since the logarithm and the exponential function are each other's inverse, we may write it as

$$e^{x_j} - \ln(\textstyle\sum_{i=1}^{K} e^{x_i})$$

The problematic part is the logarithm of the sum of exponentials. However, we can move e^c for any constant c outside the sum easily, that is,

$$\ln\left(\sum_i e^{x_i}\right) = \ln\left(e^c \sum_i e^{x_i - c}\right) = c + \ln\left(\sum_i e^{x_i - c}\right).$$

We need to find a good c, and we choose $c = \max_i x_i$ since e^{x_i} is the dominant term in the sum. We are less concerned with values being inadvertently rounded to zero since that does not change the value of the sum significantly.

Implement Softmax Regression with Gradient Descent

Follow the steps from the logistic regression implementation and implement softmax regression. Train and run the classifier on the OCR training data and report the error. Include regularization again and model selection again and see if that helps.

Report Follow the same procedure as for logistic regression. At the end, compare the two methods. Which is better in terms of classifier performance, runtime, etc.? You have to hand in code just like for logistic regression. To be precise:

Code Requirements

Cost Function and Gradient Function named `soft_cost` that takes 3 inputs, data X, target y, parameters θ. X is a $n \times (d+1)$ matrix, y is an $n \times K$ matrix and θ is a $(d+1) \times K$ matrix. The function must return the cost function for Softmax Regression and the gradient ∇E_{in} when applied to data X, y with parameters θ.

Gradient Descent Algorithm Function `soft_run` in file that takes three arguments X, y, θ as above and runs gradient descent and return the best parameters found on the data. The algorithm should be optimized for the auDigits training set.

Best Parameters Found Besides your Python file, save the best parameters you have found in a file with `np.savez("params.npz", theta=best_theta)` and hand it in. The parameters should be optimized for the auDigits data set.