Exercise 1

Deadline: 28.04.2021, 4:00 pm

This exercise focuses on training a binary classifier on handwritten numbers from the scikit-learn digits dataset (similar to the MNIST datatset). You will implement optimization methods for minimizing the logistic regression objective function and compare their convergence rates.

Regulations

Please hand-in your solution as a jupyter notebook `logistic-regression.ipynb`, accompanied with exported PDF `logistic-regression.pdf`. Zip both files into a single archive with naming convention (sorted alphabetically by last names)

`lastname1-firstname1_lastname2-firstname2_exercise01.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_exercise01.zip`

and upload it to MAMPF before the given deadline. We will give zero points if your zip-file does not conform to the naming convention.

Reminder: Regularized Logistic Regression

The lecture introduced logistic regression as a binary classifier that assigns a label $y_i \in \{-1, 1\}$ to a sample $X_i \in \mathbb{R}^D$ (a row vector) in terms of a winner-takes-all decision

$$\widehat{y} = \text{argmax}_{k\in\{-1,1\}}\, P(y = k|X_i, \widehat{\beta}) = \begin{cases} 1 & \text{if } X_i\widehat{\beta} > 0 \\ -1 & \text{if } X_i\widehat{\beta} \leq 0 \end{cases}$$

with parameter $\widehat{\beta} \in \mathbb{R}^D$ (a column vector). We assume that the intercept has been absorbed into $\widehat{\beta}$ by appending a column of ones to the feature matrix X. The posterior probability is modelled by the sigmoid function

$$P(y = 1|X_i, \widehat{\beta}) = \sigma(X_i\widehat{\beta}) = \frac{1}{1 + \exp(-X_i\widehat{\beta})} \qquad P(y = -1|X_i, \widehat{\beta}) = 1 - \sigma(X_i\widehat{\beta}) = \sigma(-X_i\widehat{\beta})$$

and the optimal parameter vector $\widehat{\beta}$ is found by minimizing the regularized logistic loss $\mathcal{L}(\beta)$ over the training set

$$\widehat{\beta} = \text{argmin}_\beta\, \mathcal{L}(\beta) \quad \text{with} \quad \mathcal{L}(\beta) = \frac{\beta^T\beta}{2\lambda} + \frac{1}{N}\sum_i \log(1 + \exp(-y_i X_i\beta))$$

where λ is the inverse regularization parameter (i.e. small values of λ lead to strong regularization). In the following we will apply different training methods to find the optimal $\widehat{\beta}$. Since this optimization problem is convex, all algorithms should return the same solution (within numerical precision). If this is not the case, there is a bug in your code or in the formulas given here (please report these, if you find any).

1 Loading the Dataset (2 points)

The following lines show how to load the digits dataset from sklearn and how to extract the necessary data (install sklearn using conda, as in the previous semester):

```
from sklearn.datasets import load_digits

digits = load_digits()

print digits.keys()

data         = digits["data"]
images       = digits["images"]
target       = digits["target"]
target_names = digits["target_names"]

import numpy as np
print data.dtype
```

Note that each row of the feature matrix `data` is a vectorized (flattened) version of the corresponding image. We restrict the dataset to two similar looking numbers to form a binary classification problem. Therefore, create a reduced feature matrix X by slicing the dataset such that it only contains the instances where `target[i]` is 3 or 8. In addition, append a column of ones to the resulting matrix to take care of the intercept. Create a matching vector of ground-truth labels with

$$y_i = \begin{cases} 1 & \text{if target}[i] = 3 \\ -1 & \text{if target}[i] = 8 \end{cases}$$

The slicing reduces the dataset to N samples and you should have the matching numpy arrays $X \in \mathbb{R}^{N \times D}$ and $y \in \mathbb{R}^N$ with $D = 65$ and $N = 357$.

1.1 Classification with sklearn (6 Points)

At first, solve the problem using sklearn's predifined class `LogisticRegression`[1]. In particular, use cross-validation to determine the optimal regularization parameter λ (called `C` in sklearn). Report cross-validation accuracies of different λ and select the best one for subsequent experiments.

1.2 Optimization Methods (20 Points)

Now, we want to compare various optimization methods that were introduced in the lecture. As a prerequisite, implement some basic functions:

- `sigmoid(z)` applies the sigmoid function to every element of the array z with $\sigma(z) = \frac{1}{1+e^{-z}}$.
- `gradient(beta, X, y)` returns the gradient of the loss function $\nabla_\beta \mathcal{L}(\beta)$ for the given arguments, where `X` and `y` are arbitrary subsets of the training data. That is, they contain a single random instance for stochastic gradient descent, B random instances for mini-batch gradient descent, and the full training set for standard gradient descent.
- `predict(beta, X)` returns a vector with one class label y_i for each row X_i in X.
- `zero_one_loss(y_prediction, y_truth)` counts the number of wrongly classified samples.

Note that these functions should support vectorized operation: If the argument is a vector or matrix, they are applied to each element or row individually (as appropriate), just like built-in numpy functions such as `numpy.exp(X)`. To increase the speed of your code you should avoid unnecessary `for`-loops. Visit `http://ufldl.stanford.edu/wiki/index.php/Logistic_Regression_Vectorization_Example` for suggestions on how to use matrix multiplication in order to calculate the gradient.

Use your helper functions to implement the following optimization methods (with hard-coded regularization parameter as determined in task 1.1). Here, τ_t is the learning rate in iteration t and $l^{(t)}$ denotes the corresponding loss gradient, as returned from `gradient(beta, X, y)` with suitable arguments:

[1] `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

gradient descent

$\beta^{(t+1)} = \beta^{(t)} - \tau_t \, l^{(t)}$

(compute the gradient from the full training set)

SG (stochastic gradient)

$\beta^{(t+1)} = \beta^{(t)} - \tau_t \, l^{(t)}$

(compute the gradient from a single random instance for every t)

SG minibatch

$\beta^{(t+1)} = \beta^{(t)} - \tau_t \, l^{(t)}$

(compute the gradient from a random mini-batch of size B for every t)

SG momentum

$g^{(t+1)} = \mu \, g^{(t)} + (1-\mu) \, l^{(t)}$

$\beta^{(t+1)} = \beta^{(t)} - \tau_t \, g^{(t+1)}$

(compute the gradient from a single random instance for every t)

ADAM

$g^{(t+1)} = \mu_1 \, g^{(t)} + (1-\mu_1) \, l^{(t)} \qquad \mu_1 \approx 0.9$

$q^{(t+1)} = \mu_2 \, q^{(t)} + (1-\mu_2) \, \left(l^{(t)}\right)^2 \qquad \mu_2 \approx 0.999$

$\tilde{g}^{(t+1)} = \frac{g^{(t+1)}}{1-\mu_1^{t+1}} \qquad \tilde{q}^{(t+1)} = \frac{q^{(t+1)}}{1-\mu_2^{t+1}} \qquad$ (bias correction)

$\beta^{(t+1)} = \beta^{(t)} - \frac{\tau}{\sqrt{\tilde{q}^{(t+1)}}+\epsilon} \, \tilde{g}^{(t+1)} \qquad \tau \approx 10^{-4}, \epsilon \approx 10^{-8}$

(compute the gradient from a single random instance for every t)

stochastic average gradient

initialization:

$\forall i : g_i^{(\text{stored})} = -y_i X_i^T \cdot \sigma\left(-y_i X_i \beta^{(0)}\right)$

$g^{(0)} = \frac{1}{N} \sum_i g_i^{(\text{stored})}$

iteration (choose a random instance i for every t):

$g_i^{(t)} = -y_i X_i^T \cdot \sigma\left(-y_i X_i \beta^{(t)}\right)$ (update gradient of instance i)

$g^{(t+1)} = g^{(t)} + \frac{1}{N}\left(g_i^{(t)} - g_i^{(\text{stored})}\right)$

$g_i^{(\text{stored})} = g_i^{(t)}$

$\beta^{(t+1)} = \beta^{(t)}\left(1 - \frac{\tau_t}{\lambda}\right) - \tau_t \, g^{(t+1)}$

dual coordinate ascent

initialization:

$\alpha^{(0)} = \text{random}(0,1)$

$\beta^{(0)} = \frac{\lambda}{N} \sum_i \alpha_i^{(0)} y_i X_i^T$

iteration (choose a random instance i for every t):

$f' = y_i X_i \beta^{(t)} + \log \frac{\alpha_i^{(t)}}{1-\alpha_i^{(t)}}, \quad f'' = \frac{\lambda}{N} X_i X_i^T + \frac{1}{\alpha_i^{(t)}\left(1-\alpha_i^{(t)}\right)}$

$\alpha_i^{(t+1)} = \underbrace{\alpha_i^{(t)} - \frac{f'}{f''}}_{\text{clip to } [\epsilon, 1-\epsilon]} \qquad$ (update α_i such that $0 < \alpha_i < 1$)

$\beta^{(t+1)} = \beta^{(t)} + \frac{\lambda}{N} y_i X_i^T \left(\alpha_i^{(t+1)} - \alpha_i^{(t)}\right)$

Newton/Raphson

$z^{(t)} = X\beta^{(t)}$ (scores) $\quad \tilde{y}^{(t)} = \frac{y}{\sigma(y \, z^{(t)})}$ (weighted labels)

$W_{ii}^{(t)} = \frac{\lambda}{N} \sigma(z_i^{(t)}) \sigma(-z_i^{(t)}) \quad$ ($W^{(t)}$ is a $N \times N$ diagonal matrix of weights)

$\beta^{(t+1)} = (I + X^T W^{(t)} X)^{-1} X^T W^{(t)} (z^{(t)} + \tilde{y}^{(t)})$

For each method define a function that makes m update steps and returns $\beta^{(m)}$. Sample random instances/mini-batches either with or without replacement and give a (one sentence) explanation why you selected your method. Note that stochastic gradient methods need a decreasing learning rate in order to converge. Use the scaling

$$\tau_t = \frac{\tau_0}{1 + \gamma t}$$

Please make X, y, $\beta^{(0)}$, τ, τ_0, γ and μ available as function parameters.

1.3 Comparison(12 Points)

For your analysis split the data into training and test set with

```
from sklearn import cross_validation
X, X_test, y, y_test = cross_validation.train_test_split(X, y, test_size=0.3,
    random_state=0)
```

Initialize $\beta^{(0)} = 0$ and $g^{(0)} = 0$, excpet in dual coordinate ascent whose initial gues must be non-zero. Here you can use random numbers: $\alpha^{(0)} =$ `np.random.uniform(size=N)`.

Learning Rate

Use the cross validation tool from sklearn to find good values for the learning rate and free parameters of each optimization algorithm. You can generate multiple training and test sets with

```
from sklearn import cross_validation
kf = cross_validation.KFold(y.shape[0], n_folds=10)
for train_index, validation_index in kf:
    X_train, X_validation = X[train_index], X[validation_index]
    y_train, y_validation = y[train_index], y[validation_index]
```

For the cross validation set the parameters sequentially to the following values:

τ or τ_0	0.001	0.01	0.1
μ	0.1	0.2	0.5
γ	0.0001	0.001	0.01

Use 150 iterations for all stochastic methods and 10 otherwise. Calculate the total error by summing over the `zero_one_loss(y_validation, predict(beta, X_validation))`, where β was trained on `(X_train,y_train)`. Finally, list the parameters with the lowest total error.

Speed

We would like to compare the convergence speeds for the algorithms. Unfortunately we can not simply use a timing tool since this is highly dependent on the implementation of each algorithm. Alternatively, we can estimate times T by multiplying the number of iterations with the computational complexity of one iteration. The complexities per iteration are $O(ND)$ for gradient descent, $O(BD)$ for SG minibatch, $O(D)$ for all other stochastic processes, and $O(ND^2)$ for Newton/Raphson. Run every algorithm with different numbers of iterations (with the parameters found in 1.3) and compute T, training- and test error for every run. Create graphs where you plot both errors as functions of T. You may want to use logarithmic axes. Which of the algorithms is the fastest to converge? Are there qualitative differences between methods?