4. Multinomial (Softmax) Regression and Gradient Descent

Extension Note: Project 2 due date has been extended by 2 days to **July 18 23:59UTC** (Note the UTC time zone).

Daniel suggests that instead of building ten models, we can expand a single logistic regression model into a multinomial regression and solve it with similar gradient descent algorithm.

The main function which you will call to run the code you will implement in this section is `run_softmax_on_MNIST` in main.py (already implemented). In the appendix at the bottom of this page, we describe a number of the methods that are already implemented for you in `softmax.py` that will be useful.

In order for the regression to work, you will need to implement three methods. Below we describe what the functions should do. We have included some test cases in `test.py` to help you verify that the methods you have implemented are behaving sensibly.

You will be working in the file `part1/softmax.py` in this problem

Computing Probabilities for Softmax

5.0/5.0 points (graded)

Write a function `compute_probabilities` that computes, for each data point $x^{(i)}$, the probability that $x^{(i)}$ is labeled as j for $j = 0, 1, \ldots, k-1$.

The softmax function h for a particular vector x requires computing

$$h(x) = \frac{1}{\sum_{j=0}^{k-1} e^{\theta_j \cdot x/\tau}} \begin{bmatrix} e^{\theta_0 \cdot x/\tau} \\ e^{\theta_1 \cdot x/\tau} \\ \vdots \\ e^{\theta_{k-1} \cdot x/\tau} \end{bmatrix},$$

where $\tau > 0$ is the **temperature parameter** . When computing the output probabilities (they should always be in the range $[0,1]$), the terms $e^{\theta_j \cdot x/\tau}$ may be very large or very small, due to the use of the exponential function. This can cause numerical or overflow errors. To deal with this, we can simply subtract some fixed amount c from each exponent to keep the resulting number from getting too large. Since

$$\begin{aligned} h(x) &= \frac{e^{-c}}{e^{-c}\sum_{j=0}^{k-1} e^{\theta_j \cdot x/\tau}} \begin{bmatrix} e^{\theta_0 \cdot x/\tau} \\ e^{\theta_1 \cdot x/\tau} \\ \vdots \\ e^{\theta_{k-1} \cdot x/\tau} \end{bmatrix} \\ &= \frac{1}{\sum_{j=0}^{k-1} e^{[\theta_j \cdot x/\tau] - c}} \begin{bmatrix} e^{[\theta_0 \cdot x/\tau] - c} \\ e^{[\theta_1 \cdot x/\tau] - c} \\ \vdots \\ e^{[\theta_{k-1} \cdot x/\tau] - c} \end{bmatrix}, \end{aligned}$$

subtracting some fixed amount c from each exponent will not change the final probabilities. A suitable choice for this fixed amount is $c = \max_j \theta_j \cdot x/\tau$.

Correctio note: We have made corrections to the indices in the equations above. The newly **corrected ones** are shown in blue.

Reminder: You can implement this function locally first, and run `python test.py` in your project1 directory to validate basic functionality before checking against the online grader here.

Available Functions: You have access to the NumPy python library as `np`; No need to import anything.

```
def compute_probabilities(X, theta, temp_parameter):
    """
    Computes, for each datapoint X[i], the probability that X[i] is labeled as j
    for j = 0, 1, ..., k-1

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        theta - (k, d) NumPy array, where row j represents the parameters of our model for label j
        temp_parameter - the temperature parameter of softmax function (scalar)
    Returns:
        H - (k, n) NumPy array, where each entry H[j][i] is the probability that X[i] is labeled as j
    """
    c = (theta @ X.T / temp_parameter).max(0)
    softmax_H = np.e ** (theta @ X.T / temp_parameter - c) / np.sum(np.e ** (theta @ X.T / temp_parameter- c), 0)
    return softmax_H

```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
def compute_probabilities(X, theta, temp_parameter):
    """
    Computes, for each datapoint X[i], the probability that X[i] is labeled as j
    for j = 0, 1, ..., k-1

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        theta - (k, d) NumPy array, where row j represents the parameters of our model for label j
        temp_parameter - the temperature parameter of softmax function (scalar)
    Returns:
        H - (k, n) NumPy array, where each entry H[j][i] is the probability that X[i] is labeled as j
    """
    itemp = 1 / temp_parameter
    dot_products = itemp * theta.dot(X.T)
    max_of_columns = dot_products.max(axis=0)
    shifted_dot_products = dot_products - max_of_columns
    exponentiated = np.exp(shifted_dot_products)
    col_sums = exponentiated.sum(axis=0)
    return exponentiated / col_sums
```

Test results

See full output

CORRECT

See full output

Submit

You have used 1 of 20 attempts

Answers are displayed within the problem

Cost Function

5.0/5.0 points (graded)

Write a function `compute_cost_function` that computes the total cost over every data point.

The cost function $J(\theta)$ is given by: (Use natural log)

$$J(\theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=0}^{k-1}[[y^{(i)} == j]]\log\frac{e^{\theta_j\cdot x^{(i)}/\tau}}{\sum_{l=0}^{k-1}e^{\theta_l\cdot x^{(i)}/\tau}}\right] + \frac{\lambda}{2}\sum_{j=0}^{k-1}\sum_{i=0}^{d-1}\theta_{ji}^2$$

Correction note: We have made corrections to the indices in the equations above. The newly **corrected ones** are shown in blue.

Available Functions: You have access to the NumPy python library as `np` and the previous function as `compute_probabilities`

Correction to test.py: The version of the project archive earlier than July 1 contains an error in part1/test.py. You could either download the corrected mnist.tar.gz version now, or correct Lines 176 and 180 test.py as follows:

```
@@ -173,11 +173,11 @@ def check_compute_cost_function():
173         ex_name = "Compute cost function"
174         n, d, k = 3, 5, 7
175         X = np.arange(0, n * d).reshape(n, d)
    -       Y = np.arange(0, n).reshape(n, 1)
176 +       Y = np.arange(0, n)
177         zeros = np.zeros((k, d))
178         temp = 0.2
179         lambda_factor = 0.5
    -       exp_res = 5.83773044716594
180 +       exp_res = 1.9459101490553135
181         if check_real(
182                 ex_name, softmax.compute_cost_function,
183                 exp_res, X, Y, zeros, lambda_factor, temp):
```

```
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns
        c - the cost value (scalar)
    """
    n = X.shape[0]
    k = theta.shape[0]
    correct = np.zeros([k,n])
    for i in range(n):
        correct[Y[i]][i] = 1
    clipped_probs = np.clip(compute_probabilities(X, theta, temp_parameter), 1e-15, 1-1e-15)
    loss = -1/n * np.sum(correct * np.log(clipped_probs))
    regularization = lambda_factor/2 * np.linalg.norm(theta)**2
    return loss + regularization
```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
def compute_cost_function(X, Y, theta, lambda_factor, temp_parameter):
    """
    Computes the total cost over every datapoint.

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for each
            data point
        theta - (k, d) NumPy array, where row j represents the parameters of our
                model for label j
        lambda_factor - the regularization constant (scalar)
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns
        c - the cost value (scalar)
    """
    N = X.shape[0]
    probabilities = compute_probabilities(X, theta, temp_parameter)
    selected_probabilities = np.choose(Y, probabilities)
    non_regulizing_cost = np.sum(np.log(selected_probabilities))
    non_regulizing_cost *= -1 / N
    regulizing_cost = np.sum(np.square(theta))
    regulizing_cost *= lambda_factor / 2.0
    return non_regulizing_cost + regulizing_cost
```

Test results

See full output

CORRECT

See full output

Submit You have used 2 of 20 attempts

Answers are displayed within the problem

Gradient Descent

5.0/5.0 points (graded)

Solution to this problem available before due date: The function `run_gradient_descent_iteration` is necessary for the rest of the project. Hence, once you have either submitted the correct function or finished your attempts for this problem, the solution to this function will be available. We are sorry we acted this late, but hope that it will still help.

Now, in order to run the gradient descent algorithm to minimize the cost function, we need to take the derivative of $J(\theta)$ wrt a particular θ_m. Notice that within $J(\theta)$, we have:

$$\frac{e^{\theta_j \cdot x^{(i)}/\tau}}{\sum_{l=0}^{k-1} e^{\theta_l \cdot x^{(i)}/\tau}} = p(y^{(i)} = j|x^{(i)}, \theta)$$

so we first compute: $\frac{\partial p(y^{(i)}=j|x^{(i)},\theta)}{\partial \theta_m}$,

when $m = j$,

$$\frac{\partial p(y^{(i)} = j|x^{(i)}, \theta)}{\partial \theta_m} = \frac{x^{(i)}}{\tau} p(y^{(i)} = m|x^{(i)}, \theta)\left[1 - p(y^{(i)} = m|x^{(i)}, \theta)\right]$$

when $m \neq j$,

$$\frac{\partial p(y^{(i)} = j|x^{(i)}, \theta)}{\partial \theta_m} = -\frac{x^{(i)}}{\tau} p(y^{(i)} = m|x^{(i)}, \theta)\, p(y^{(i)} = j|x^{(i)}, \theta)$$

Now we compute

$$\begin{aligned}
\frac{\partial}{\partial \theta_m}\left[\sum_{j=0}^{k-1} [[y^{(i)} == j]] \log \frac{e^{\theta_j \cdot x^{(i)}/\tau}}{\sum_{l=0}^{k-1} e^{\theta_l \cdot x^{(i)}/\tau}}\right] &= \sum_{j=0, j\neq m}^{k-1}\left[[[y^{(i)} == j]]\left[-\frac{x^{(i)}}{\tau} p(y^{(i)} = m|x^{(i)}, \theta)\right]\right. \\
&\quad \left. + [[y^{(i)} == m]] \frac{x^{(i)}}{\tau}\left[1 - p(y^{(i)} = m|x^{(i)}, \theta)\right]\right] \\
&= \frac{x^{(i)}}{\tau}\left[[[y^{(i)} == m]] - p(y^{(i)} = m|x^{(i)}, \theta) \sum_{j=0}^{k-1} [[y^{(i)} == j]]\right] \\
&= \frac{x^{(i)}}{\tau}\left[[[y^{(i)} == m]] - p(y^{(i)} = m|x^{(i)}, \theta)\right]
\end{aligned}$$

Plug this into the derivatite of $J(\theta)$, we have

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_m} &= \frac{\partial}{\partial \theta_m}\left[-\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{j=0}^{k-1} [[y^{(i)} == j]] \log p(y^{(i)} = j|x^{(i)}, \theta)\right] + \frac{\lambda}{2}\sum_{j=0}^{k-1}\sum_{i=0}^{d-1} \theta_{ji}^2\right] \\
&= -\frac{1}{\tau n}\sum_{i=1}^{n}\left[x^{(i)}\left([[y^{(i)} == m]] - p(y^{(i)} = m|x^{(i)}, \theta)\right)\right] + \lambda \theta_m
\end{aligned}$$

To run gradient descent, we will update θ at each step with $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$, where α is the learning rate.

Write a function `run_gradient_descent_iteration` that runs one step of the gradient descent algorithm.

Available Functions: You have access to the NumPy python library as `np`, `compute_probabilities` which you previously implemented and `scipy.sparse` as `sparse`

You should use `sparse.coo_martix` so that your function can handle larger matrices efficiently (and not time out for the online graders). The sparse matrix representation can handle sparse matrices efficiently.

Hint

This is how to use `scipy` 's `sparse.coo_matrix` function to create a sparse matrix of 0's and 1's:

```
M = sparse.coo_matrix(([1]*n, (Y, range(n))), shape=(k,n)).toarray()
```

This will create a normal numpy array with 1s and 0s.

On larger inputs (i.e. MNIST), this is 10x faster than using a naive for loop. (See example code if interested).

Note: As a personal challenge, try to see if you can use special numpy functions to add 1 in-place. This would be even faster.

```
import time
import numpy as np
import scipy.sparse as sparse

ITER = 100
K = 10
N = 10000

def naive(indices, k):
    mat = [[1 if i == j else 0 for j in range(k)] for i in indices]
    return np.array(mat).T


def with_sparse(indices, k):
    n = len(indices)
    M = sparse.coo_matrix(([1]*n, (Y, range(n))), shape=(k,n)).toarray()
    return M


Y = np.random.randint(0, K, size=N)

t0 = time.time()
for i in range(ITER):
    naive(Y, K)
print(time.time() - t0)


t0 = time.time()
for i in range(ITER):
    with_sparse(Y, K)
print(time.time() - t0)
```

Hide

```
        lambda_factor - the regularization constant (scalar)
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        theta - (k, d) NumPy array that is the final value of parameters theta
    """
    n = X.shape[0]
    k = theta.shape[0]
    correct = np.zeros([k,n])
    for i in range(n):
        correct[Y[i]][i] = 1
    clipped_probs = np.clip(compute_probabilities(X, theta, temp_parameter), 1e-15, 1-1e-15)
    jacobian = -1/(n*temp_parameter) * (correct - clipped_probs)@X + lambda_factor*theta
    return theta - alpha*jacobian

```

Press ESC then TAB or click outside of the code editor to exit

Correct

```
def run_gradient_descent_iteration(X, Y, theta, alpha, lambda_factor, temp_parameter):
    """
    Runs one step of batch gradient descent

    Args:
        X - (n, d) NumPy array (n datapoints each with d features)
        Y - (n, ) NumPy array containing the labels (a number from 0-9) for each
            data point
        theta - (k, d) NumPy array, where row j represents the parameters of our
                model for label j
        alpha - the learning rate (scalar)
        lambda_factor - the regularization constant (scalar)
        temp_parameter - the temperature parameter of softmax function (scalar)

    Returns:
        theta - (k, d) NumPy array that is the final value of parameters theta
    """
    itemp=1./temp_parameter
    num_examples = X.shape[0]
    num_labels = theta.shape[0]
    probabilities = compute_probabilities(X, theta, temp_parameter)
    # M[i][j] = 1 if y^(j) = i and 0 otherwise.
    M = sparse.coo_matrix(([1]*num_examples, (Y,range(num_examples))), shape=(num_labels,num_examples)).toarray()
    non_regularized_gradient = np.dot(M-probabilities, X)
    non_regularized_gradient *= -itemp/num_examples
    return theta - alpha * (non_regularized_gradient + lambda_factor * theta)
```

Test results

See full output

CORRECT

See full output

Submit You have used 1 of 20 attempts

Answers are displayed within the problem

Test Error on Softmax Regression

1.0/1.0 point (graded)

Finally, report the final test error by running the `main.py` file, using the temperature parameter $\tau = 1$. If you have implemented everything correctly, the error on the test set should be around 0.1, which implies the linear softmax regression model is able to recognize MNIST digits with around 90 percent accuracy.

Note: For this project we will be looking at the error rate defined as the fraction of labels that don't match the target labels, also known as the "gold labels" or ground truth. (In other contexts, you might want to consider other performance measures such as precision and recall, which we have not discussed in this class.

Please enter the **test error** of your Softmax algorithm (copy the output from the `main.py` run).

0.1005 ✔ **Answer:** 0.1005

Submit You have used 1 of 20 attempts

Answers are displayed within the problem

Discussion

Show Discussion

Topic: Unit 2 Nonlinear Classification, Linear regression, Collaborative Filtering (2 weeks):Project 2: Digit recognition (Part 1) / 4. Multinomial (Softmax) Regression and Gradient Descent