Q2.2 logistic.py

October 17, 2019

0.1 Q2.2 logistic.py

[1]:
```
""" Methods for doing logistic regression."""

import numpy as np
from utils import sigmoid

def logistic_predict(weights, data):
    """
    Compute the probabilities predicted by the logistic classifier.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to the bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
    Outputs:
        y:          :N x 1 vector of probabilities. This is the output of the␣
↪classifier.
    """
    # TODO: Finish this function
    N = data.shape[0]
    # append vector contains only 1s
    data = np.concatenate((data, np.full((N, 1), 1)), axis=1)

    # predict the output given weights
    y = sigmoid(np.matmul(data, weights))

    return y

def evaluate(targets, y):
    """
    Compute evaluation metrics.
    Inputs:
```

```
        targets : N x 1 vector of targets.
        y       : N x 1 vector of probabilities.
    Outputs:
        ce           : (scalar) Cross entropy. CE(p, q) = E_p[-log q]. Here we␣
↪want to compute CE(targets, y)
        frac_correct : (scalar) Fraction of inputs classified correctly.
    """
    # TODO: Finish this function
    # Squeeze 2d column or row vector into 1d array
    targets = np.squeeze(targets)
    y = np.squeeze(y)
    # Calculate cross entropy loss
    ce = -(np.dot(targets, np.log(y + 1e-8)) + np.dot(1 - targets, np.log(1 - y␣
↪+ 1e-8)))

    # Calculate the percentage of correct prediction with 0.5 as threshold
    correct_num = np.sum(np.absolute(y - targets) <= .5)
    frac_correct = correct_num / float(targets.shape[0])

    return ce, frac_correct

def logistic(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to␣
↪weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
        targets:    N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:       The sum of the loss over all data points. This is the␣
↪objective that we want to minimize.
        df:      (M+1) x 1 vector of derivative of f w.r.t. weights.
        y:       N x 1 vector of probabilities.
    """
    # TODO: Finish this function
    no_penalize = hyperparameters.copy()
    no_penalize["weight_regularization"] = 0
```

```python
    return logistic_pen(weights, data, targets, no_penalize)


def logistic_pen(weights, data, targets, hyperparameters):
    """
    Calculate negative log likelihood and its derivatives with respect to␣
 ↪weights.
    Also return the predictions.

    Note: N is the number of examples and
          M is the number of features per example.

    Inputs:
        weights:    (M+1) x 1 vector of weights, where the last element
                    corresponds to bias (intercepts).
        data:       N x M data matrix where each row corresponds
                    to one data point.
        targets:    N x 1 vector of targets class probabilities.
        hyperparameters: The hyperparameters dictionary.

    Outputs:
        f:             The sum of the loss over all data points. This is the␣
 ↪objective that we want to minimize.
        df:            (M+1) x 1 vector of derivative of f w.r.t. weights.
    """

    # TODO: Finish this function
    y = logistic_predict(weights, data)

    # Calculate the difference between the prediction and true value
    diff = targets - y

    # Compute gradient of dL/dw, and change to column vector
    df = (np.matmul(diff.T, data)).T

    # Add derivative of bias to the last column
    df = np.append(df, [[np.sum(diff)]], axis=0)
    df = -df

    # Derived the regulizer and add to the derivative
    df[:-1] += hyperparameters["weight_regularization"] * weights[:-1] * target.
 ↪shape[0]
    # Compute cross entropy loss
    f, _ = evaluate(targets, y)

    # Include the regulizer
```

```
    f += (hyperparameters["weight_regularization"] / 2) * np.sum(weights**2) *␣
↪target.shape[0]
    return f, df, y
```

[ ]:

```

```