CPSC 340 Assignment 4 (due Wednesday, Mar 6 at 11:55pm)

Student name: José Abraham Torres Juárez
Student id: 79507828

Instructions

Rubric: {mechanics:5}

IMPORTANT!!! Before proceeding, please carefully read the general homework instructions at `https://www.cs.ubc.ca/~fwood/CS340/homework/`. The above 5 points are for following the submission instructions. You can ignore the words "mechanics", "reasoning", etc.

We use blue to highlight the deliverables that you must answer/do/submit with the assignment.

1 Convex Functions

Rubric: {reasoning:5}

Recall that convex loss functions are typically easier to minimize than non-convex functions, so it's important to be able to identify whether a function is convex.

Show that the following functions are convex:

1. $f(w) = \alpha w^2 - \beta w + \gamma$ with $w \in \mathbb{R}, \alpha \geq 0, \beta \in \mathbb{R}, \gamma \in \mathbb{R}$ (1D quadratic).

$$f'(w) = 2\alpha w - \beta$$
$$f''(w) = 2\alpha$$

   Since $\alpha \geq 0$ the second derivative is non-negative, thus the $f(w)$ is convex

2. $f(w) = -\log(\alpha w)$ with $\alpha > 0$ and $w > 0$ ("negative logarithm")

$$f'(w) = -\frac{\cancel{\alpha}}{\cancel{\alpha} w} = -\frac{1}{w}$$
$$f''(w) = \frac{1}{w^2}$$

   Since $w > 0$ the second derivative is non-negative, thus the $f(w)$ is convex

3. $f(w) = \|Xw - y\|_1 + \frac{\lambda}{2}\|w\|_1$ with $w \in \mathbb{R}^d, \lambda \geq 0$ (L1-regularized robust regression).

$$f(w) = \sum_{i=1}^{n} |w^T X_i - y_i| + \frac{\lambda}{2} \sum_{i=1}^{d} |w_i|$$

We know that a sum of two convex functions is also convex, let $g(w) = |w^T X_i - y_i|$ and $h(w_i) = |w_i|$

$$g(w) = |w^T X_i - y_i|$$
$$g'(w) = X_i$$
$$g''(w) = 0$$

So $g(w)$ is convex

$$h(w_i) = |w_i|$$
$$h'(w) = 1$$
$$h''(w) = 0$$

So $h(w)$ is convex, thus $f(w)$ is convex because it is the result of the sum of multiple convex functions.

4. $f(w) = \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i))$ with $w \in \mathbb{R}^d$ (logistic regression).
   let $z = -y_i w^T x_i$, and let $g(w) = \log(1 + \exp(z))$

$$g'(w) = \frac{\exp(z)}{1 + \exp(z)} = \frac{1}{1 + \exp(-z)}$$

$$g''(w) = \frac{\exp(-z)}{(1 + \exp(-z))^2}$$

   $g''(w)$ will always be > 0, because exp() of something is never gonna be negative, thus $f(w)$ is convex because it is a summatory $g(w)$ that is always convex

5. $f(w) = \sum_{i=1}^{n} [\max\{0, |w^T x_i - y_i|\} - \epsilon] + \frac{\lambda}{2}\|w\|_2^2$ with $w \in \mathbb{R}^d, \epsilon \geq 0, \lambda \geq 0$ (support vector regression).
   A max function of convex functions will also be convex. We are taking the max of two functions,

   (a) $g() = 0$

   (b) $h(w) = |w^T x_i - y_i|$

   We know that g will be convex. For $h(w)$

$$h'(w) = x_i$$
$$h''(w) = 1$$

   So $h(w)$ is convex because its second derivative is non-negative.
   Thus the whole max function is convex, and also the summatory of this max functions. This summatory is adding another function $l(w) = \frac{\lambda}{2}\|w\|_2^2$

$$l'(w) = \frac{\cancel{2}\lambda}{\cancel{2}} w = \lambda w$$

$$l'(w) = \frac{\cancel{2}\lambda}{\cancel{2}} w = \lambda w$$

   So $l(w)$ is also convex which will make $f(w)$ a convex function.

General hint: for the first two you can check that the second derivative is non-negative since they are one-dimensional. For the last 3 you'll have to use some of the results regarding how combining convex functions can yield convex functions which can be found in the lecture slides.

Hint for part 4 (logistic regression): this function may seem non-convex since it contains $\log(z)$ and log is concave, but there is a flaw in that reasoning: for example $\log(\exp(z)) = z$ is convex despite containing a log. To show convexity, you can reduce the problem to showing that $\log(1 + \exp(z))$ is convex, which can be done by computing the second derivative. It may simplify matters to note that $\frac{\exp(z)}{1+\exp(z)} = \frac{1}{1+\exp(-z)}$.

2 Logistic Regression with Sparse Regularization

If you run `python main.py -q 2`, it will:

1. Load a binary classification dataset containing a training and a validation set.
2. 'Standardize' the columns of X and add a bias variable (in *utils.load_dataset*).
3. Apply the same transformation to $Xvalidate$ (in *utils.load_dataset*).
4. Fit a logistic regression model.
5. Report the number of features selected by the model (number of non-zero regression weights).
6. Report the error on the validation set.

Logistic regression does reasonably well on this dataset, but it uses all the features (even though only the prime-numbered features are relevant) and the validation error is above the minimum achievable for this model (which is 1 percent, if you have enough data and know which features are relevant). In this question, you will modify this demo to use different forms of regularization to improve on these aspects.

Note: your results may vary a bit depending on versions of Python and its libraries.

2.1 L2-Regularization

Rubric: {code:2}

Make a new class, *logRegL2*, that takes an input parameter λ and fits a logistic regression model with L2-regularization. Specifically, while *logReg* computes w by minimizing

$$f(w) = \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)),$$

your new function *logRegL2* should compute w by minimizing

$$f(w) = \sum_{i=1}^{n} \left[\log(1 + \exp(-y_i w^T x_i))\right] + \frac{\lambda}{2}\|w\|^2.$$

Hand in your updated code. Using this new code with $\lambda = 1$, report how the following quantities change: the training error, the validation error, the number of features used, and the number of gradient descent iterations.

Note: as you may have noticed, `lambda` is a special keyword in Python and therefore we can't use it as a variable name. As an alternative we humbly suggest `lammy`, which is what Mike's niece calls her stuffed animal toy lamb. However, you are free to deviate from this suggestion. In fact, as of Python 3 one can now use actual greek letters as variable names, like the λ symbol. But, depending on your text editor, it may be annoying to input this symbol.

By using the L2 regularization the training error was **0.002**, and the validation error **0.074** it was decreased by **0.01** in comparison to the `logReg` that doesn't make regularization. The number of gradient descent iterations was **30**.

Below is the implementation of the *logRegL2* class.

```
class logRegL2(logReg):
    def __init__(self, lammy=1.0, verbose=0, maxEvals=100):
        self.lammy = lammy
        self.verbose = verbose
        self.maxEvals = maxEvals
        self.bias = True

    def funObj(self, w, X, y):
        yXw = y * X.dot(w)

        # Calculate the function value
        f = np.sum(np.log(1. + np.exp(-yXw))) + (self.lammy/2.0)*w.dot(w)

        # Calculate the gradient value
        res = - y / (1. + np.exp(yXw))
        g = X.T.dot(res) + self.lammy*w
        return f, g
```

2.2 L1-Regularization

Rubric: {code:3}

Make a new class, *logRegL1*, that takes an input parameter λ and fits a logistic regression model with L1-regularization,

$$f(w) = \sum_{i=1}^{n} \left[\log(1 + \exp(-y_i w^T x_i))\right] + \lambda \|w\|_1.$$

Hand in your updated code. Using this new code with $\lambda = 1$, report how the following quantities change: the training error, the validation error, the number of features used, and the number of gradient descent iterations.

You should use the function *minimizers.findMinL1*, which implements a proximal-gradient method to minimize the sum of a differentiable function g and $\lambda\|w\|_1$,

$$f(w) = g(w) + \lambda\|w\|_1.$$

This function has a similar interface to *findMin*, **EXCEPT** that (a) you only pass in the the function/gradient of the differentiable part, g, rather than the whole function f; and (b) you need to provide the value λ. To reiterate, your `funObj` **should not contain the L1 regularization term**; rather it should only implement the function value and gradient for the training error term. The reason is that the optimizer handles the non-smooth L1 regularization term in a specialized way (beyond the scope of CPSC 340).

Using L1 regularization we obtained a training error of **0.000** and a validation error of **0.052**. So we got much more accuracy in the validation set than the L2 regularized and non regularized logistic regression.

The number of gradient descent iterations was **47**.

Below is the implementation of the *logRegL1* class.

```
class logRegL1(logReg):
    def __init__(self, L1_lambda=1.0, verbose=0, maxEvals=100):
        self.L1_lambda = L1_lambda
        self.verbose = verbose
        self.maxEvals = maxEvals
```

```
        self.bias = True

    def fit(self,X, y):
        n, d = X.shape

        # Initial guess
        self.w = np.zeros(d)
        utils.check_gradient(self, X, y)
        (self.w, f) = findMin.findMinL1(self.funObj, self.w, self.L1_lambda,
                                        self.maxEvals, X, y, verbose=self.verbose)
```

2.3 L0-Regularization

Rubric: {code:4}

The class *logRegL0* contains part of the code needed to implement the *forward selection* algorithm, which approximates the solution with L0-regularization,

$$f(w) = \sum_{i=1}^{n} \left[\log(1 + \exp(-y_i w^T x_i))\right] + \lambda \|w\|_0.$$

The `for` loop in this function is missing the part where we fit the model using the subset *selected_new*, then compute the score and updates the *minLoss/bestFeature*. Modify the `for` loop in this code so that it fits the model using only the features *selected_new*, computes the score above using these features, and updates the *minLoss/bestFeature* variables. Hand in your updated code. Using this new code with $\lambda = 1$, report the training error, validation error, and number of features selected.

Note that the code differs a bit from what we discussed in class, since we assume that the first feature is the bias variable and assume that the bias variable is always included. Also, note that for this particular case using the L0-norm with $\lambda = 1$ is equivalent to what is known as the Akaike Information Criterion (AIC) for variable selection.

Also note that, for numerical reasons, your answers may vary depending on exactly what system and package versions you are using. That is fine.

Using L0 regularization we obtained a training error of **0.000** and a validation error of **0.018**. This Logistic regressor got much more accuracy in the validation set than the L1, L2 and non regularized logistic regression.

This is because the model does feature selection considering just **24** out of **101**, so by doing this it probably reduces the noise that extra features will cause in the regressor. Below is the implementation of the *logRegL0* class.

```
class logRegL0(logReg):
    # L0 Regularized Logistic Regression
    def __init__(self, L0_lambda=1.0, verbose=2, maxEvals=400):
        self.verbose = verbose
        self.L0_lambda = L0_lambda
        self.maxEvals = maxEvals

    def funObj(self, w, X, y):
        yXw = y * X.dot(w)

        # Calculate the function value
        f = np.sum(np.log(1. + np.exp(-yXw)))  + (w != 0).sum()*self.L0_lambda
```

```
        # Calculate the gradient value
        res = - y / (1. + np.exp(yXw))
        g = X.T.dot(res)

        return f, g

    def fit(self, X, y):
        n, d = X.shape
        minimize = lambda ind: findMin.findMin(self.funObj,
                                               np.zeros(len(ind)),
                                               self.maxEvals,
                                               X[:, ind], y, verbose=0)
        selected = set()
        selected.add(0)
        minLoss = np.inf
        oldLoss = 0
        bestFeature = -1

        while minLoss != oldLoss:
            oldLoss = minLoss
            print("Epoch %d " % len(selected))
            print("Selected feature: %d" % (bestFeature))
            print("Min Loss: %.3f\n" % minLoss)

            for i in range(d):
                if i in selected:
                    continue

                selected_new = selected | {i} # tentatively add feature "i" to the seected set

                # TODO for Q2.3: Fit the model with 'i' added to the features,
                # then compute the loss and update the minLoss/bestFeature
                loss = minimize(list(selected_new))[1]
                if(loss < minLoss):
                    minLoss = loss
                    bestFeature = i

            selected.add(bestFeature)

        self.w = np.zeros(d)
        self.w[list(selected)], _ = minimize(list(selected))
```

2.4 Discussion

Rubric: {reasoning:2}

In a short paragraph, briefly discuss your results from the above. How do the different forms of regularization compare with each other? Can you provide some intuition for your results? No need to write a long essay, please!

The main difference between L1 and L2 regularization is that using L1 results in a w with some features in 0, which will reduce the noise that those (un-relevant) features that are introducing to our regression. So this feature selection ends in an improvement on our validation error. We noticed this in the 2.2 problem which selected **71** features and had **0.052** validation error. Moreover by using *forward selection* algorithm on problem 2.3 we selected only **24** features and obtained a much better, **0.018**, validation error.

2.5 Comparison with scikit-learn

Rubric: {reasoning:1}

Compare your results (training error, validation error, number of nonzero weights) for L2 and L1 regularization with scikit-learn's LogisticRegression. Use the `penalty` parameter to specify the type of regularization. The parameter `C` corresponds to $\frac{1}{\lambda}$, so if you had $\lambda = 1$ then use `C=1` (which happens to be the default anyway). You should set `fit_intercept` to `False` since we've already added the column of ones to X and thus there's no need to explicitly fit an intercept parameter. After you've trained the model, you can access the weights with `model.coef_`.

For the L2 scikit-learn's LogisticRegression the training error was **0.002**, and the validation error a **0.074**. And for the L1 scikit-learn's LogisticRegression the training error was **0.000**, and the validation error a **0.052**. Which are the exact errors that we got on our implementations of L2 and L1 logReg respectively.

2.6 $L\frac{1}{2}$ regularization

Rubric: {reasoning:4}

Previously we've considered L2 and L1 regularization which use the L2 and L1 norms respectively. Now consider least squares linear regression with "$L\frac{1}{2}$ regularization" (in quotation marks because the "$L\frac{1}{2}$ norm" is not a true norm):

$$f(w) = \frac{1}{2}\sum_{i=1}^{n}(w^T x_i - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j|^{1/2}\,.$$

Let's consider the case of $d = 1$ and assume there is no intercept term being used, so the loss simplifies to

$$f(w) = \frac{1}{2}\sum_{i=1}^{n}(wx_i - y_i)^2 + \lambda \sqrt{|w|}\,.$$

Finally, let's assume $n = 2$ where our 2 data points are $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (0, 1)$.

1. Plug in the data set values and write the loss in its simplified form, without a summation.

$$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2}(0-1)^2 + \lambda\sqrt{|w|}$$

$$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2} + \lambda\sqrt{|w|}$$

2. If $\lambda = 0$, what is the solution, i.e. $\arg\min_w f(w)$?

$$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2} + 0\sqrt{|w|}$$

$$f'(w) = w - 2$$

$$0 = w - 2 \quad \boxed{w = 2}$$

3. If $\lambda \to \infty$, what is the solution, i.e., $\arg\min_w f(w)$?

$$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2} + \infty\sqrt{|w|}$$

By considering the above function, as λ grows to ∞ the regularization term will become bigger and bigger, the minimum of the function will be acquired by setting $\boxed{w = 0}$ that will make the regularization term to be 0 and leave use with $f(0) = \frac{1}{2}(0-2)^2 + \frac{1}{2} + 0$

4. Plot $f(w)$ when $\lambda = 1$. What is $\arg\min_w f(w)$ when $\lambda = 1$? Answer to one decimal place if appropriate.

$$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2} + \sqrt{|w|}$$

$$f'(w) = w - 2 + \frac{1}{2\sqrt{|w|}}$$

$$2 = w + \frac{1}{2\sqrt{|w|}} \quad \boxed{w \approx 1.6}$$

5. Plot $f(w)$ when $\lambda = 10$. What is $\arg\min_w f(w)$ when $\lambda = 10$? Answer to one decimal place if appropriate.

$$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2} + 10\sqrt{|w|}$$

The minimum value of $f(w)$ we can get needs to cancel the regularization term because $\lambda = 10$, with a non-zero w would be pretty big. Thus if we set $\boxed{w = 0}$ we get the minimum of $f(w)$.

6. Does $L\frac{1}{2}$ regularization behave more like L1 regularization or L2 regularization when it comes to performing feature selection? Briefly justify your answer.

$L\frac{1}{2}$ behaves more like L1 regularization, by setting a large λ we got a $w = 0$

7. Is least squares with $L\frac{1}{2}$ regularization a convex optimization problem? Briefly justify your answer.

It is non-convex because the second derivative of the regularization term is negative.

$$g(w) = \lambda\sqrt{|w|}$$

$$g'(w) = \frac{\lambda}{2\sqrt{|w|}}$$

$$g''(w) = -\frac{\lambda}{4\sqrt{|w|}}$$

3 Multi-Class Logistic

If you run `python main.py -q 3` the code loads a multi-class classification dataset with $y_i \in \{0, 1, 2, 3, 4\}$ and fits a 'one-vs-all' classification model using least squares, then reports the validation error and shows a plot of the data/classifier. The performance on the validation set is ok, but could be much better. For example, this classifier never even predicts that examples will be in classes 0 or 4.

3.1 Softmax Classification, toy example

Rubric: {reasoning:2}

Linear classifiers make their decisions by finding the class label c maximizing the quantity $w_c^T x_i$, so we want to train the model to make $w_{y_i}^T x_i$ larger than $w_{c'}^T x_i$ for all the classes c' that are not y_i. Here c' is a possible

label and $w_{c'}$ is row c' of W. Similarly, y_i is the training label, w_{y_i} is row y_i of W, and in this setting we are assuming a discrete label $y_i \in \{1, 2, \ldots, k\}$. Before we move on to implementing the softmax classifier to fix the issues raised in the introduction, let's work through a toy example:

Consider the dataset below, which has $n = 10$ training examples, $d = 2$ features, and $k = 3$ classes:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}, \quad y = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 2 \\ 2 \\ 3 \\ 3 \\ 3 \end{bmatrix}.$$

Suppose that you want to classify the following test example:

$$\hat{x} = \begin{bmatrix} 1 & 1 \end{bmatrix}.$$

Suppose we fit a multi-class linear classifier using the softmax loss, and we obtain the following weight matrix:

$$W = \begin{bmatrix} +2 & -1 \\ +2 & -2 \\ +3 & -1 \end{bmatrix}$$

Under this model, what class label would we assign to the test example? (Show your work.)

For determining what class to assign we would need to compute the `argmax` of $\hat{x}W^T$ this will result in a $1 \times k$ vector which we will find the index of the max value in it, and that would be the class we should pick.

$$\hat{x}W^T = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} +2 & +2 & +3 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2 \end{bmatrix}$$

$$\text{class} = argmax(\hat{x}W^T) = argmax(\begin{bmatrix} 1 & 0 & 2 \end{bmatrix}) = \boxed{3}$$

So for $\hat{x}$ we should pick the class 3

3.2 One-vs-all Logistic Regression

Rubric: {code:2}

Using the squared error on this problem hurts performance because it has 'bad errors' (the model gets penalized if it classifies examples 'too correctly'). Write a new class, *logLinearClassifier*, that replaces the squared loss in the one-vs-all model with the logistic loss. Hand in the code and report the validation error.

The One-vs-all Logistic Regression obtained a training error of **0.084** and a validation error of **0.070**. This regressor had only **15** non-zero weights.

Below is the implementation of the *logLinearClassifier*

```
class logLinearClassifier(logReg):

    def funObj(self, w, X, y):
```

```
        yXw = y * X.dot(w)

        # Calculate the function value
        f = np.sum(np.log(1. + np.exp(-yXw)))

        # Calculate the gradient value
        res = - y / (1. + np.exp(yXw))
        g = X.T.dot(res)

        return f, g

    def fit(self, X, y):
        n, d = X.shape
        self.n_classes = np.unique(y).size

        # Initial guess
        self.W = np.zeros((self.n_classes,d))

        for i in range(self.n_classes):
            ytmp = y.copy().astype(float)
            ytmp[y==i] = 1
            ytmp[y!=i] = -1

            # solve the normal equations
            # with a bit of regularization for numerical reasons
            (self.W[i], f) = findMin.findMin(self.funObj, self.W[i],
                                             self.maxEvals, X, ytmp, verbose=self.verbose)

    def predict(self, X):
        return np.argmax(X@self.W.T, axis=1)
```

3.3 Softmax Classifier Gradient

Rubric: {reasoning:5}

Using a one-vs-all classifier can hurt performance because the classifiers are fit independently, so there is no attempt to calibrate the columns of the matrix W. As we discussed in lecture, an alternative to this independent model is to use the softmax loss, which is given by

$$f(W) = \sum_{i=1}^{n}\left[-w_{y_i}^T x_i + \log\left(\sum_{c'=1}^{k}\exp(w_{c'}^T x_i)\right)\right],$$

Show that the partial derivatives of this function, which make up its gradient, are given by the following expression:

$$\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n} x_{ij}[p(y_i = c \mid W, x_i) - I(y_i = c)],$$

where...

- $I(y_i = c)$ is the indicator function (it is 1 when $y_i = c$ and 0 otherwise)

- $p(y_i = c \mid W, x_i)$ is the predicted probability of example i being class c, defined as

$$p(y_i = c \mid W, x_i) = \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}$$

$f(W)$ is a composition of to functions, let

$$g(W) = \sum_{i=1}^{n} -w_{y_i}^T x_i$$

$$h(W) = \sum_{i=1}^{n} \log\left(\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)\right)$$

$$\frac{\partial g}{\partial W_{cj}} = \sum_{i=1}^{n} -x_{ij} I(y_i = c)$$

In the previous derivative when $c \neq y_i$ the partial derivative should be 0, so we are using the, indicator function previously

$$\frac{\partial h}{\partial W_{cj}} = \sum_{i=1}^{n} \frac{x_{ij} \exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}$$

$$\text{So: } \frac{\partial f}{\partial W_{cj}} = \frac{\partial g}{\partial W_{cj}} + \frac{\partial h}{\partial W_{cj}}$$

$$\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n} -x_{ij} I(y_i = c) + \sum_{i=1}^{n} \frac{x_{ij} \exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}$$

$$\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n} \left[-x_{ij} I(y_i = c) + \frac{x_{ij} \exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}\right]$$

$$\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n} x_{ij} \left[-I(y_i = c) + \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}\right]$$

$$\text{let } p(y_i = c \mid W, x_i) = \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}$$

So the partial derivatives of the Softmax loss is the following:

$$\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n} x_{ij} \left[p(y_i = c \mid W, x_i) - I(y_i = c)\right]$$

3.4 Softmax Classifier Implementation

Rubric: {code:5}

Make a new class, *softmaxClassifier*, which fits W using the softmax loss from the previous section instead of fitting k independent classifiers. Hand in the code and report the validation error.

Hint: you may want to use `utils.check_gradient` to check that your implementation of the gradient is correct.

Hint: with softmax classification, our parameters live in a matrix W instead of a vector w. However, most optimization routines like `scipy.optimize.minimize`, or the optimization code we provide to you, are set up to optimize with respect to a vector of parameters. The standard approach is to "flatten" the matrix W into a vector (of length kd, in this case) before passing it into the optimizer. On the other hand, it's inconvenient to work with the flattened form everywhere in the code; intuitively, we think of it as a matrix W and our code will be more readable if the data structure reflects our thinking. Thus, the approach we recommend is to reshape the parameters back and forth as needed. The `funObj` function is directly communicating with the optimization code and thus will need to take in a vector. At the top of `funObj` you can immediately reshape the incoming vector of parameters into a $k \times d$ matrix using `np.reshape`. You can then compute the gradient using sane, readable code with the W matrix inside `funObj`. You'll end up with a gradient that's also a matrix: one partial derivative per element of W. Right at the end of `funObj`, you can flatten this gradient matrix into a vector using `grad.flatten()`. If you do this, the optimizer will be sending in a vector of parameters to `funObj`, and receiving a gradient vector back out, which is the interface it wants – and your `funObj` code will be much more readable, too. You may need to do a bit more reshaping elsewhere, but this is the key piece.

The Softmax Classifier obtained a training error of **0.000** and a validation error of **0.008**. And it had only **15** non-zero weights.

Below is the implementation of the *softmaxClassifier*

```
class softmaxClassifier(logReg):

    def funObj(self, w, X, y):
        k, d  = self.W_shape
        n, d  = X.shape
        W = w.reshape((k, d))

        # Calculate the function value
        f = 0
        sumExps = np.zeros(n)
        for i in range(n):
            Wyi = W[y[i]]
            f -= Wyi.T.dot(X[i])
            for cprime in range(k):
                sumExps[i] += np.exp(W[cprime].T.dot(X[i]))
            f += np.log(sumExps[i])

        # Calculate the gradient value
        g = np.zeros((k, d))
        I = lambda yi, c: 1 if yi == c else 0
        for c in range(k):
            for j in range(d):
                sumN = 0
                for i in range(n):
                    Iyi = I(y[i], c)
                    xij = X[i,j]
                    pYi = np.exp(W[c].T.dot(X[i]))/sumExps[i]
                    sumN += xij*(pYi - Iyi)

                g[c,j] = sumN

        return f, g.flatten()

    def fit(self, X, y):
```

```
        n, d = X.shape
        self.n_classes = np.unique(y).size

        # Initial guess
        self.W = np.zeros((self.n_classes,d))
        self.W_shape = self.W.shape

        self.w = self.W.flatten()
        utils.check_gradient(self, X, y)
        (flatteneddW, f) = findMin.findMin(self.funObj, self.W.flatten(),
                                           self.maxEvals, X, y, verbose=self.verbose)
        self.W = flatteneddW.reshape(self.W_shape)

    def predict(self, X):
        return np.argmax(X@self.W.T, axis=1)
```

3.5 Comparison with scikit-learn, again

Rubric: {reasoning:1}

Compare your results (training error and validation error for both one-vs-all and softmax) with scikit-learn's `LogisticRegression`, which can also handle multi-class problems. One-vs-all is the default; for softmax, set `multi_class='multinomial'`. For the softmax case, you'll also need to change the solver. You can use `solver='lbfgs'`. Since your comparison code above isn't using regularization, set `C` very large to effectively disable regularization. Again, set `fit_intercept` to `False` for the same reason as above (there is already a column of 1's added to the data set).
Again for both implemententations One-vs-All and Softmax our results matched the Scikit-learn's `LogisticRegression` class results.

For the One-vs-All the `LogisticRegression` model was created with the followin parameters:

1. $C = 9999$,
2. $multi\_class =' ovr'$,
3. $solver =' liblinear'$,
4. $fit\_intercept = False$

For the Softmax the `LogisticRegression` model was created with the followin parameters:

1. $C = 9999$,
2. $multi\_class =' multinomial'$,
3. $solver =' lbfgs'$,
4. $fit\_intercept = False$

3.6 Cost of Multinomial Logistic Regression

Rubric: {reasoning:2}

Assume that we have

- n training examples.

- d features.
- k classes.
- t testing examples.
- T iterations of gradient descent for training.

Also assume that we take X and form new features Z using Gaussian RBFs as a non-linear feature transformation.

1. In $O()$ notation, what is the cost of training the softmax classifier with gradient descent?
   Z is an $n \times n$ matrix so we need n^2 for generate every value Z_{ij}. And the computation of every Z_{ij} value we are calculating $\exp(-\frac{\epsilon^2}{2\sigma^2})$ where ϵ is the distance between training examples X_i and X_j which will take d time to compute. Therefore the creation of Z will have $O(n^2 d)$ time.

   After creating Z we need to use gradient descent with softmax, which will normally take $nd + nkd$ time but since we are using Z to calculate this, we have $d = n$ so the time would be $n^2 + kn^2$. We need to do calculate this function T times until convergence, so the total running time of gradient descent with softmax would be $O(Tn^2 + Tkn^2)$.

   By adding the two running times, the creation of Z $O(n^2 d)$ and gradient descent with softmax $O(Tn^2 + Tkn^2)$, we get that the total of creating Z and train on it would be $O(Tn^2 + Tkn^2 + n^2 d)$
2. What is the cost of classifying the t test examples? For the classification we would also need to transform our test data using Gaussian RBF transformation this would generate a $n \times t$ matrix $\tilde{Z}$. We can generate this by calculating the Gaussian RBF function $\exp(-\frac{\epsilon^2}{2\sigma^2})$ ϵ where is the distance between training examples $\hat{X}_i$ and X_j. This would take $O(ntd)$ time.

   The classification of the test data needs to compute $W\tilde{Z}$ where W is a $k \times n$ matrix so this will take $O(nkt)$ time, and will result in a $k \times t$ matrix that we need to calculate the `argmax` of each column to get the correct class for each t test example, the `argmax` will take k time and we will do this t times so $O(kt)$.

   The whole process of classifying t test examples is has a running time of $O(ntd + nkt + kt)$

Hint: you'll need to take into account the cost of forming the basis at training (Z) and test ($\tilde{Z}$) time. It will be helpful to think of the dimensions of all the various matrices.

4 Very-Short Answer Questions

Rubric: {reasoning:12}

1. Suppose that a client wants you to identify the set of "relevant" factors that help prediction. Why shouldn't you promise them that you can do this?
   Because this factors would depend on the training data, we are not sure that we have enough data or enough features to really distinguish the "relevant" factors from the "non-relevant" ones.
2. Consider performing feature selection by measuring the "mutual information" between each column of X and the target label y, and selecting the features whose mutual information is above a certain threshold (meaning that the features provides a sufficient number of "bits" that help in predicting the label values). Without delving into any details about mutual information, what is a potential problem with this approach?

This approach is more likely to "memorize" the data, it sounds like the aproach of decision trees, if so we might get a pretty small training error, but probably a big validation error.

3. What is a setting where you would use the L1-loss, and what is a setting where you would use L1-regularization?
   When we don't want to be affected by outliers we would use L1-loss, on the other hand when we are willing to just consider the man most "relevant" features on the dataset for our predictions we would use L1-regularization.

4. Among L0-regularization, L1-regularization, and L2-regularization: which yield convex objectives? Which yield unique solutions? Which yield sparse solutions?
   L0-regularization yields convex objectives L1-regularization yields sparse solutions L2-regularization yields unique solution

5. What is the effect of λ in L1-regularization on the sparsity level of the solution? What is the effect of λ on the two parts of the fundamental trade-off?
   As λ grows it would make the solution more sparse, also it makes the approximation error smaller, but the validation error is bigger.

6. Suppose you have a feature selection method that tends not generate false positives but has many false negatives (it misses relevant variables). Describe an ensemble method for feature selection that could improve the performance of this method.
   We could use a random forest approach, by bootstrapping several training sets and then run the same feature selection method on all and average the output of all for the prediction.

7. Suppose a binary classification dataset has 3 features. If this dataset is "linearly separable", what does this precisely mean in three-dimensional space?
   The data would be separated by a plane.

8. When searching for a good w for a linear classifier, why do we use the logistic loss instead of just minimizing the number of classification errors?
   Because the linear classifier uses continous data.

9. What are "support vectors" and what's special about them?
   The support vectors are the training data points that are closest to the decision hyperplane. If the training data we are using is linearly separable these "support vectors" are "supporting" the hyperplane that perfectly separates the training points of one class from another one.

10. What is a disadvantage of using the perceptron algorithm to fit a linear classifier?
    The perceptron can't handle non-linearly separable data. So if the training data that we are using is not linearly separable it would output a very bad model.

11. Why we would use a multi-class SVM loss instead of using binary SVMs in a one-vs-all framework?
    Because the one-vs-all approach is generating k independent classifiers so the decision boundaries will try to isolate each class from the rest of the training set. So on the whole they might not model all the training data in a proper way, we would just have k decision hyperplanes separating the data.

    On the other hand a multi-class SVM will try to find the class that is more likely to be. This allow us to have much more complex decision boundaries because it is considering all the classes at the training time.

12. How does the hyper-parameter σ affect the shape of the Gaussian RBFs bumps? How does it affect the fundamental tradeoff?
    As σ grows it will flatten the Gaussian RBFs bumps, making it more likely to have samples on the tails of the Gaussian RBF bump.
    The value of σ should reduce the approximation error, but increase the validation error.