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

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).
   $f'(w) = 2\alpha w - \beta$
   $f''(w) = 2\alpha \geq 0$
2. $f(w) = -\log(\alpha w)$ with $\alpha > 0$ and $w > 0$ ("negative logarithm")
   $f'(w) = -\frac{\alpha}{\alpha w}$
   $f''(w) = \frac{\alpha^2}{(\alpha w)^2} \geq 0$
3. $f(w) = \|Xw - y\|_1 + \frac{\lambda}{2}\|w\|_1$ with $w \in \mathbb{R}^d, \lambda \geq 0$ (L1-regularized robust regression).
   Norms are convex functions and $Xw - y$ is a linear function. The composition of a convex function with a linear function is convex, so $\|Xw - y\|_1$ is convex. Also, $\frac{\lambda}{2}\|w\|_1$ is convex because it is a convex function multiplied by a non-negative constant. Since the sum of convex functions is convex, $f(w)$ is convex.
4. $f(w) = \sum_{i=1}^n \log(1 + \exp(-y_i w^T x_i))$ with $w \in \mathbb{R}^d$ (logistic regression).
   Show that $f(z) = \log(1 + \exp(z))$ is convex:
   $f'(z) = \frac{\exp(z)}{1+\exp(z)} = \frac{1}{1+\exp(-z)}$
   $f''(z) = \frac{\exp(-z)}{(1+\exp(-z))^2} \geq 0$
   so $f(z) = \log(1 + \exp(z))$ is convex.
   Since $-y_i w^T x_i$ is a linear function and the composition of a linear function with convex function is convex, $\log(1 + \exp(-y_i w^T x_i))$ is convex. Since the sum of convex functions is convex, $f(w)$ is 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).
   Since $w^T x_i - y_i$ is a linear function and $f(x) = |x|$ is convex, $|w^T x_i - y_i|$ is convex as it is the composition of a convex function with a linear function. Note that 0 and ϵ are convex as their second derivative is 0. Since the max of convex functions is convex, $\max\{0, |w^T x_i - y_i|\}$ is convex. Thus,

$[\max\{0, |w^T x_i - y_i|\} - \epsilon]$ is convex as it is the sum of convex functions. Also, $\frac{\lambda}{2}\|w\|_2^2$ is convex because it is a convex function (squared norm is convex) multiplied by a non-negative constant. Since the sum of convex functions is convex, $f(w)$ is convex.

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.
logRegL2 Training error 0.002
logRegL2 Validation error 0.074
number of feature used: 101
number of gradient descent iterations: 36

```
class logRegL2:
    # Logistic Regression
    def __init__(self, lammy, verbose=0, maxEvals=100):
        self.verbose = verbose
        self.maxEvals = maxEvals
        self.bias = True
        self.lammy = lammy

    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) * (w.T@w)

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

    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.findMin(self.funObj, self.w,
                                      self.maxEvals, X, y, verbose=self.verbose)
    def predict(self, X):
        return np.sign(X@self.w)
```

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).
logRegL1 Training error 0.000
logRegL1 Validation error 0.052
number of feature used: 71
number of gradient descent iterations: 78

```
class logRegL1:
    # Logistic Regression
    def __init__(self, L1_lambda, verbose=0, maxEvals=100):
        self.verbose = verbose
        self.maxEvals = maxEvals
        self.bias = True
        self.L1_lambda = L1_lambda

    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

        # 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)
    def predict(self, X):
        return np.sign(X@self.w)
```

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.
Training error 0.000
Validation error 0.032
number of features selected: 24

```
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 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 selected set
                # TODO for Q2.3: Fit the model with 'i' added to the features,
                # then compute the loss and update the minLoss/bestFeature
                print(minimize(list(selected_new)))
                (self.w, f) = minimize(list(selected_new))
                f += self.L0_lambda * len(selected_new)
                if f < minLoss:
                    minLoss = f
                    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!
According to the data above, the training error is 0 for both L0 and L1, but non-zero for L2. L0 has the lowest validation error while L2 has the highest validation error. L0 uses the lowest number of features while L2 uses the highest number of features.

L1 and L2 are convex and they improve our test errors. L1 and L0 encourages elements of 'w' to be exactly 0, that is, they give sparsity but L2 doesn't give sparsity. L2 focuses on decreasing the largest(making w_j similar) whereas L1 focuses on decreasing all w_j until they are 0. However, L0 requires $O(d^2)$ run time which makes it slower than L1 and L2.

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 L2:
Training error 0.002
Validation error 0.074
number of nonZeros: 101

For L1:
Training error 0.000
Validation error 0.052
number of nonZeros: 71

For both L1 and L2, the results are the same for both our implementation and sklearn's implementation.

```
elif question == "2.5":
data = utils.load_dataset("logisticData")
XBin, yBin = data['X'], data['y']
XBinValid, yBinValid = data['Xvalid'], data['yvalid']

# TODO
model = LogisticRegression(penalty='l2', C=1.0, fit_intercept=False, solver='liblinear')
model.fit(XBin, yBin)
print("\nTraining error %.3f" % utils.classification_error(model.predict(XBin), yBin))
print("Validation error %.3f" % utils.classification_error(model.predict(XBinValid), yBinValid)
print("# nonZeros: %d" % (model.coef_ != 0).sum())

model = LogisticRegression(penalty='l1', C=1.0, fit_intercept=False, solver='liblinear')
model.fit(XBin, yBin)
print("\nTraining error %.3f" % utils.classification_error(model.predict(XBin), yBin))
print("Validation error %.3f" % utils.classification_error(model.predict(XBinValid), yBinValid)
print("# nonZeros: %d" % (model.coef_ != 0).sum())
```

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):

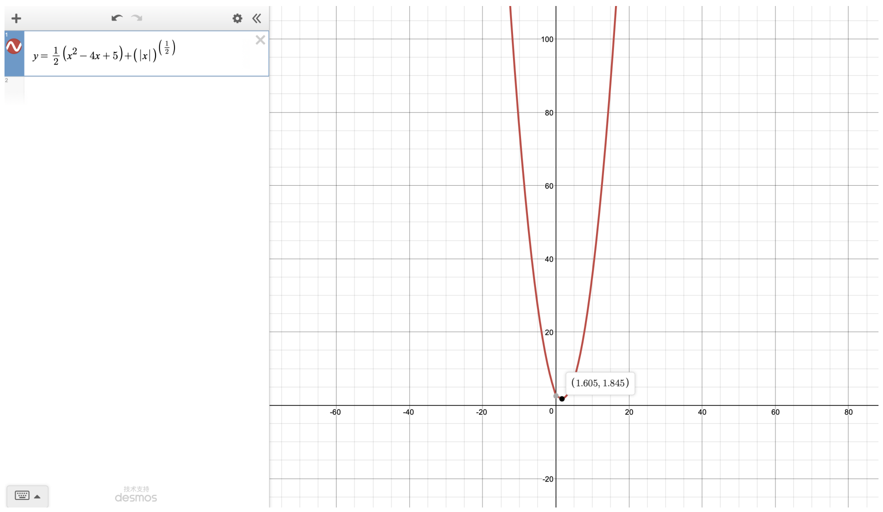
$$f(w) = \frac{1}{2}\sum_{i=1}^{n}(w^T x_i - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j|^{1/2}.$$

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

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

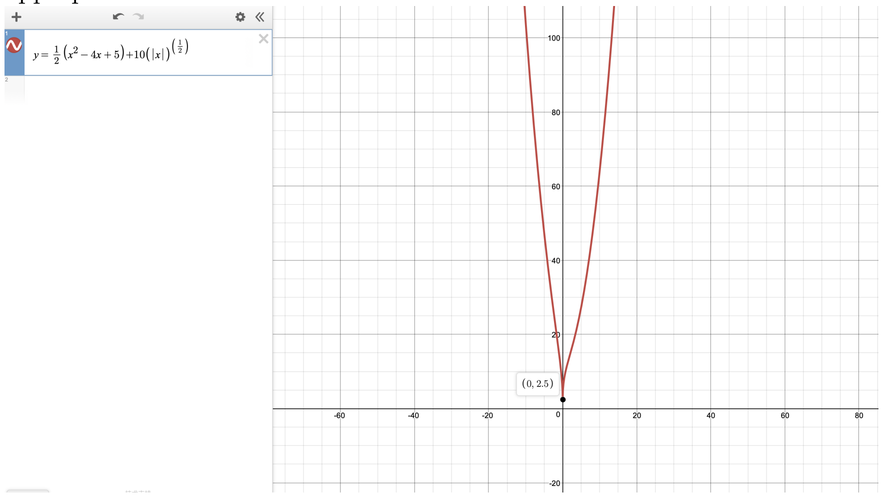
Finally, let's assume $n = 2$ where our 2 data points are $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (0, 1)$.

1. Plug in the data set values and write the loss in its simplified form, without a summation.
   $f(w) = \frac{1}{2}(w^2 - 4w + 5) + \lambda\sqrt{|w|}$
2. If $\lambda = 0$, what is the solution, i.e. $\arg\min_w f(w)$?
   when $\lambda = 0$, $f(w) = \frac{1}{2}(w^2 - 4w + 5)$
   $\nabla f(w) = w - 2 = 0$
   $\arg\min_w f(w) = 2$
3. If $\lambda \to \infty$, what is the solution, i.e., $\arg\min_w f(w)$?
   With $\lambda = \infty$, we must set $w = 0$ and not use any features
   $\arg\min_w f(w) = 0$
4. Plot $f(w)$ when $\lambda = 1$. What is $\arg\min_w f(w)$ when $\lambda = 1$? Answer to one decimal place if appropriate.



$\arg\min_w f(w) = 1.6$

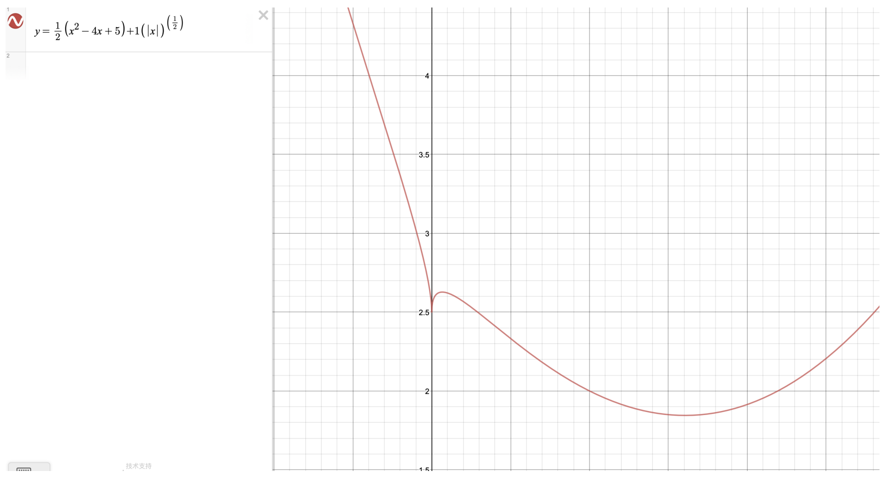
5. Plot $f(w)$ when $\lambda = 10$. What is $\arg\min_w f(w)$ when $\lambda = 10$? Answer to one decimal place if appropriate.



$\arg\min_w f(w) = 0$

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$\frac{1}{2}$ regularization behave more like L2 regularization during feature selection because the graphs above are polynomial-like rather than absolute-value like.

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



No, from the above graph we can see that it is non-convex because if i draw a line between 2 points it is possible to cross the plot.

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 $y_i \in \{1, 2, \dots, k\}$. 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.)
$w_1^T\hat{x} = 2 * 1 + (-1) * 1 = 1$
$w_2^T\hat{x} = 2 * 1 + (-2) * 1 = 0$
$w_3^T\hat{x} = 3 * 1 + (-1) * 1 = 2$
Since the linear classifier makes decision by finding the class label maximizing $w_c^T x_i$, the model assigns 3 to the test example.

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.
logLinearClassifier Training error 0.084
logLinearClassifier Validation error 0.070

```
class logLinearClassifier:
    def __init__(self, verbose=1, maxEvals=100):
        self.verbose = verbose
        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)))

        # 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):
            y_binary = y.copy().astype(float)
            y_binary[y == i] = 1
            y_binary[y != i] = -1
            (self.w[i, :], f) = findMin.findMin(self.funObj, self.w[i, :],
                                        self.maxEvals, X, y_binary, verbose=self.verbose)
    def predict(self, X):
        y_hat = X@self.w.T
        return np.argmax(y_hat, 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

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

We can represent the term inside the sum as a negative log of predicted probability of example i being class c:
$-w_{y_i}^T x_i + \log\left(\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)\right) = -log(exp(-w_{y_i}^T x_i)) - log(\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)^{-1}) = -log(\frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}) =$
$-log(p(y_i = c \mid W, x_i))$ (from bullet point 2)
Then, taking derivative with respect to W_{cj} gives:
$\frac{\partial(-log(p(y_i|W,xi)))}{\partial W_{cj}} = -I(y_i = c) * x_{ij} + \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)} * x_{ij} = -I(y_i = c) * x_{ij} + p(y_i = c \mid W, x_i) * x_{ij}$
Therefore,
$\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n} (-I(y_i = c) * x_{ij} + p(y_i = c \mid W, x_i) * x_{ij}) = \sum_{i=1}^{n} x_{ij} [p(y_i = c \mid W, x_i) - I(y_i = c)]$

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.
500 - loss: 3.890
Training error 0.000
Validation error 0.008

```
class softmaxClassifier:
    # Logistic Regression
    def __init__(self, verbose=0, maxEvals=100):
        self.verbose = verbose
        self.maxEvals = maxEvals

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

        sum2 = 0
        for i in range(n):
            sum1 = 0
            for c in range(k):
                sum1 += np.exp(w[c, :]@X[i, :].T)
            sum2 += (-1) * w[y[i], :]@X[i, :].T + np.log(sum1)

        f = sum2

        g = np.zeros((k, d))
        for c in range(k):
            for j in range(d):
                for i in range(n):
                    inner = self.getProb(w, c, X[i, :])
                    if y[i] == c:
                        inner -= 1.0
                    g[c, j] += X[i, j] * inner

        print(f, g)
        return f, g.flatten()

    def fit(self,X, y):
        n, d = X.shape
        self.n_classes = np.unique(y).size
        k = self.n_classes
        self.w = np.zeros((self.n_classes, d))
        (self.w, f) = findMin.findMin(self.funObj, self.w.flatten(),
                                        self.maxEvals, X, y, verbose=self.verbose)
        self.w = np.reshape(self.w, (k, d))

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

    def getProb(self, w, c2, x):
        sum = 0
        k = self.n_classes
        for c in range(k):
            sum += np.exp(w[c, :] @ x.T)
        return np.exp(w[c2, :] @ x.T) / sum
```

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).
One vs All:
Training error 0.084
Validation error 0.070

Softmax:
Training error 0.000
Validation error 0.016

The training and validation error for One vs All is exactly the same between our implementation and scikit-learn's implementation. While both of our implementation and scikit-learn's implementation of the Softmax classifier yield a training error of 0, the validation error is higher for Softmax in scikit-learn's implementation than ours

```
    elif question == "3.5":
        data = utils.load_dataset("multiData")
        XMulti, yMulti = data['X'], data['y']
        XMultiValid, yMultiValid = data['Xvalid'], data['yvalid']

        # TODO
        model = LogisticRegression(C=10000000, solver='liblinear')
        model.fit(XMulti, yMulti)
        print("\nTraining error %.3f" % utils.classification_error(model.predict(XMulti), yMulti))
        print("Validation error %.3f" % utils.classification_error(model.predict(XMultiValid),
        yMultiValid))
        print("# nonZeros: %d" % (model.coef_ != 0).sum())
```

```
model = LogisticRegression(multi_class='multinomial', C=10000000, fit_intercept=False,
solver='lbfgs')
model.fit(XMulti, yMulti)
print("\nTraining error %.3f" % utils.classification_error(model.predict(XMulti), yMulti))
print("Validation error %.3f" % utils.classification_error(model.predict(XMultiValid),
yMultiValid))
print("# nonZeros: %d" % (model.coef_ != 0).sum())
```

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?
   $O(Tkn^2 + n^2d)$
2. What is the cost of classifying the t test examples?
   $O(n^2k)$

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?
   It is hard to define relevant features in real world problems, each data set has distinct features.
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?
   There might be problems with collinearity, that is, there might be interactions between features. This means that if the "Tuesday" variable always equals the "taco" variable, it could say Tuesdays are relevant but tacos are not.

3. What is a setting where you would use the L1-loss, and what is a setting where you would use L1-regularization?
   I would use L1-loss if I want to be robust to outliers. I use L1-regularization if I want to be robust to irrelevant features.

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

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 the value of λ increases, the sparsity level increases. As the value of λ increases, training error goes up and approximation error goes down.

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 can create bootstrap samples from the training example and take the features selected in all bootstrap samples as relevant features.

7. Suppose a binary classification dataset has 3 features. If this dataset is "linearly separable", what does this precisely mean in three-dimensional space?
   It means that classes can be separated by a plane so a perfect linear classifier exists.

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 0-1 loss is non-convex in 'w', if no perfect classifier exist then minimizing the 0-1 loss is hard. Logistic loss is convex so it is easy to minimize.

9. What are "support vectors" and what's special about them?
   For linearly separable data, "support vectors" are the points closest to the perfect classifier with the largest margin. SVMs finds the maximum-margin classifier so the name SVM comes from "support vectors".

10. What is a disadvantage of using the perceptron algorithm to fit a linear classifier?
    The disadvantage of perceptron algorithm is that it only works for linearly separable data. It won't work if no perfect classifier exists.

11. Why we would use a multi-class SVM loss instead of using binary SVMs in a one-vs-all framework?
    A one-vs-all framework does not take into account of the interactions between the binary SVMs.

12. How does the hyper-parameter σ affect the shape of the Gaussian RBFs bumps? How does it affect the fundamental tradeoff?
    As σ increases the width of the RBF bumps increases. Since narrow bumps means the model is more complicated, as σ increases, the model gets less complicated so training error goes up and approximation error goes down.