CPSC 340 Assignment 4 (due Monday, Mar 9 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). Answer: $f''(w) = \alpha \geq 0$, so f is convex

2. $f(w) = -\log(\alpha w)$ with $\alpha > 0$ and $w > 0$ ("negative logarithm") Answer: $f''(w) = \frac{1}{w^2} > 0$ for all $w > 0$, so f 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). Answer: $w \longmapsto Xw - y$ is linear and norms are convex, so the compositions $w \longmapsto \|Xw - y\|_1$ is convex; $w \longmapsto \frac{\lambda}{2}\|w\|_1$ is convex because norms are convex and $\lambda \geq 0$. The sum of 2 convex functions is convex, hence f 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). Answer: We first show that $z \longmapsto \log(1 + \exp(z))$ is convex by taking the second derivative:

$$\frac{d^2}{dz^2}\log(1 + \exp(z)) = \frac{\exp(-z)}{(1 + \exp(-z))^2} > 0$$

for all $z \in \mathbb{R}$. For each i from 1 to n, $w \longmapsto -y_i w^T x_i$ is linear, so $w \longmapsto \log(1 + \exp(-y_i w^T x_i))$ is convex. The sum of convex functions is convex, so f 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). Answer: f can be rewritten as $f(w) = \sum_{i=1}^n \max\{0, |w^T x_i - y_i|\} + \frac{\lambda}{2}\|w\|_2^2 - n\epsilon$. Once again, $w \longmapsto w^T x_i - y_i$ is linear and norms (in this case the absolute value on $\mathbb{R}$) are convex, so $w \longmapsto |w^T x_i - y_i|$ is convex. The max of convex functions are convex and the 0 functions is convex, so $w \longmapsto \max\{0, w^T x_i - y_i\}$ is convex. $w \longmapsto \frac{\lambda}{2}\|w\|_2^2$ is convex because squared norms are convex and $\lambda \geq 0$. Finally $-n\epsilon$ is a constant function, so it is convex by the second derivative test. The sum of convex functions is convex, so f 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.

Answer:
logRegL2 Training error 0.002
logRegL2 Validation error 0.074
number of features used: 101
number of gradient descent iterations: 36

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

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

        # Calculate the function value
        f = np.sum(np.log(1. + np.exp(-yXw))) + (l/2) * np.sum(np.square(w
            ))

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

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.

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.

Answer:
logRegL1 Training error 0.000
logRegL1 Validation error 0.052
number of features used: 71
number of gradient descent iterations: 78

```
class logRegL1:
    # Logistic Regression
    def __init__(self, verbose=2, maxEvals=100, l = 0):
        self.verbose = verbose
        self.maxEvals = maxEvals
        self.bias = True
        self.l = l

    def funObj(self, w, X, y):
        l = self.l
        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.l,
        self.maxEvals, X, y, verbose=self.verbose)

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

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).

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.

Answer:
Training error 0.000
Validation error 0.020
Number of features selected: 25

```
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
            seected set

        loss = minimize(np.array(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))
```

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.

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!

Answer: The effects of L_2, L_1 and L_0 regularizations on feature selection are in increasing order (the number of features used and the validation error decrease) because we know that L_2 regularization does not incentivize feature selection since the slope of $\frac{\partial}{\partial w_i}\|w\|_2^2$ tends to 0 as w_i tends to 0. For L_1 the slope stays constant so there is incentive for w_i to be 0 to minimize f. L_0 produces the best effect on feature selection because the second summand of f, $\|w\|_0$ precisely penalizes models with too many w_i's being non-zero.

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_`.

Answer:
The results are exactly the same.
logRegL2 Training error 0.002
logRegL2 Validation error 0.074
number of features used: 101
logRegL1 Training error 0.000
logRegL1 Validation error 0.052
number of features used: 71

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):

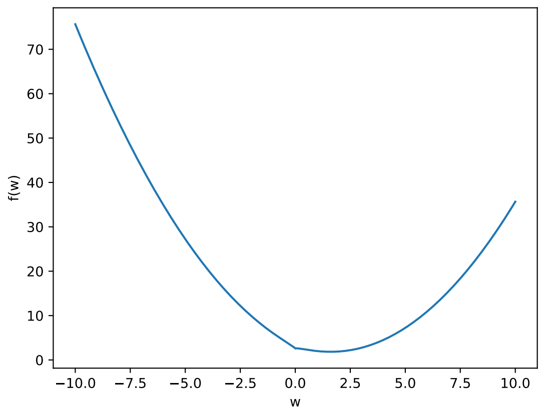
$$f(w) = \frac{1}{2}\sum_{i=1}^{n}(w^T x_i - y_i)^2 + \lambda \sum_{j=1}^{d} |w_j|^{1/2}\,.$$

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

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

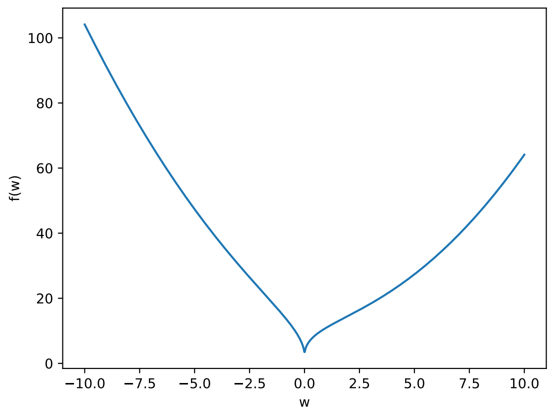
Finally, let's assume $n = 2$ where our 2 data points are $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (0, 1)$.

1. Plug in the data set values and write the loss in its simplified form, without a summation.
   Answer: $f(w) = \frac{1}{2}((w-2)^2 + 1) + \lambda\sqrt{|w|} = \frac{1}{2}w^2 - 2w + \frac{5}{2} + \lambda\sqrt{|w|}$
2. If $\lambda = 0$, what is the solution, i.e. $\arg\min_w f(w)$?
   Answer: $\arg\min_w f(w) = 2$
3. If $\lambda \to \infty$, what is the solution, i.e., $\arg\min_w f(w)$?
   Answer: $\lim_{\lambda\to\infty} \arg\min_w f(w) = 0$
4. Plot $f(w)$ when $\lambda = 1$. What is $\arg\min_w f(w)$ when $\lambda = 1$? Answer to one decimal place if appropriate.



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

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



   Answer: $\arg\min_w f(w) = 0.0$

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.
Answer: This $\frac{1}{2}$ regularization behaves more like L_1 regularization in terms of performing feature selection because for example in the 1 dimensional case above, the slope of f is large near 0 so it provides incentive for w to be closer to 0 whereas in L_2 regularization the slope is small near 0 so the incentive for w to be closer to 0 diminishes as w tends to 0.

7. Is least squares with L$\frac{1}{2}$ regularization a convex optimization problem? Briefly justify your answer.
Answer: No, because for example the function f above is not convex for λ sufficiently large ($\lambda = 10$ is depicted above).

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.)

Answer: We should assign the label 3. The rows of W are the weight vectors w_1^T, w_2^T, and w_3^T (weights of one-vs-all classifiers for each label). We compute $w_1^T \hat{x} = 1$, $w_2^T \hat{x} = 0$, and $w_3^T \hat{x} = 2$. The label 3 maximizes the value $w_c^T \hat{x}$.

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.

Answer: Validation error = 0.070. Code:

```
class logLinearClassifier:
def __init__(self, verbose=0, maxEvals=100):
self.verbose = verbose
self.maxEvals = maxEvals

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

self.W = np.zeros((self.n_classes, d))

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

model = logReg(verbose=self.verbose, maxEvals=self.maxEvals)
model.fit(X, ybin)

self.W[i] = model.w

def predict(self, X):
return np.argmax(Xself.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

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

Answer: We compute for each i:

$$\frac{\partial}{\partial W_{cj}} (-w_{y_i}^T x_i) = \frac{\partial}{\partial W_{cj}} \left(-\sum_{j=1}^d W_{y_i j} x_{ij} \right) = \begin{cases} x_{ij}, & y_i = c \\ 0, & y_i = 0 \end{cases} = -x_{ij} I(y_i = c)$$

$$\frac{\partial}{\partial W_{cj}} \left(\log \left(\sum_{c'=1}^k \exp(w_{c'}^T x_i) \right) \right) = \frac{1}{\sum_{c'=1}^k \exp(w_{c'}^T x_i)} \left(\frac{\partial}{\partial W_{cj}} \left(\sum_{c'=1}^k \exp(w_{c'}^T x_i) \right) \right) = \frac{\exp(w_{y_i}^T x_i) x_{ij}}{\sum_{c'=1}^k \exp(w_{c'}^T x_i)}$$
$$= p(y_i = c \mid W, x_i) x_{ij}$$

By linearity, it follows that

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

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.

Answer: Validation error = 0.008. Code:

```
class softmaxClassifier:
def __init__(self, verbose=0, maxEvals=100):
self.verbose = verbose
self.maxEvals = maxEvals
self.bias = True
```

```
def funObj(self, w, X, y):
n, d = X.shape
labels = np.unique(y)
k = labels.size  # num classes
W = w.reshape((k, d))

ynorm = np.array([np.argwhere(labels==c)[0][0] for c in y])

# Calculate the function value
f = np.sum(np.log(np.sum(np.exp(W.dot(x_i)))) for x_i in X)\
- np.sum(np.multiply(X,[W[y_i] for y_i in ynorm]))

# Calculate the gradient value
D = np.sum([np.exp(W.dot(x_i)) for x_i in X], axis=1)
p = np.array([np.divide(np.exp(X.dot(wc)),D) for wc in W])
I = np.array([ynorm == c for c in range(k)], dtype=float)
pI = np.add(p, -I)
g = pI  X

return f, g.flatten()

def fit(self, X, y):
n, d = X.shape
k = np.unique(y).size  # num classes

# Initial guess
self.w = np.zeros(d * k)
utils.check_gradient(self, X, y)
(self.w, f) = findMin.findMin(self.funObj, self.w,
self.maxEvals, X, y, verbose=self.verbose)
self.W = self.w.reshape(k, d)

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).

Answer:

```
logLinearClassifier Training error 0.084
logLinearClassifier Validation error 0.070
softmaxClassifier Training error 0.000
softmaxClassifier Validation error 0.008
```

```
scikitOneVsAll Training error 0.004
scikitOneVsAll Validation error 0.022
scikitSoftmax Training error 0.004
scikitSoftmax Validation error 0.022
```

Input parameters were as follows:

`scikitOneVsAll: multi_class="auto", solver="lbfgs", C=100.0, fit_intercept=False`

`scikitSoftmax: multi_class="multinomial", solver="lbfgs", C=100.0, 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?
   Answer: We first consider using the softmax classifier to fit X itself. Each step of gradient descent evaluates f and ∇f, and then computes a new value for W. The computation of W is an $O(dk)$ operation since W is a flattened vector representing the $k \times d$ weight matrix. The evaluation of f is $O(ndk)$: To evaluate the expression $w_c^T x_i$ for any c and i is $O(d)$ since w has length d. The first term can therefore be evaluated in $O(d)$ time, and the log term can be evaluated in $O(dk)$ time. There are n terms in the summation, which gives $O(ndk)$ time in total.

   The evaluation of ∇f is $O(ndk)$: In the code in part 3.4, we see that $\nabla f = PX$ where P is the $k \times n$ matrix called `pI` in part 3.4, namely $P_{c,i} = p(y_i = c \mid W, x_i) - I(y_i = c)$. The computation of P is $O(ndk)$: $I(y_i = c)$ can be computed in constant time and $\exp(w_c^T x_i)$ can be computed in $O(d)$ time. This gives $O(ndk)$ time for the computation of P since the denominator $\sum_{c'=1}^{k} \exp(w_{c'}^T x_i$ can be then computed in $O(k)$ time given $\exp(w_c^T x_i)$ has already been computed for all i and c. Multiplication of P and X is $O(ndk)$ as well so altogether, evaluating ∇f is $O(ndk)$.

   Altogether we see that training the softmax classifier with gradient descent is $O(Tndk)$. If we use a Gaussian RBF transformation, we have to compute Z, which is $O(n^2)$. Then we apply softmax classification on Z, which is $O(Tn^2k)$ since the number of features of Z is the number of training examples.

2. What is the cost of classifying the t test examples?
   Answer: To classify one test example $\hat{x}$, we label $\hat{x}$ with the label c which maximizes $w_c^T \hat{x}$. Each dot product $w_c^T \hat{x}$ is $O(d)$ and there are k features, so labelling $\hat{x}$ is $O(dk)$.

   Now suppose we have the Gaussian RBF transformation. Then we first compute the transformed features of $\hat{x}$ by computing the length n vector $\hat{z} = [g(\|x_i - \hat{x}\|)]_i$. This is an $O(n)$ computation. Then, we fit $\hat{z}$ with the Gaussian model, which is an $O(nk)$ operation. Altogether, it takes $O(tnk)$ to classify t examples.

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? Answer: Feature engineering requires a lot of domain knowledge because it is highly dependent on the context. If we only have general knowledge and no domain knowledge in the industry that the client is working in it is best to not give advice on what features we should use for our model.
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? Answer: It can be that we are selecting many features that are dependent on one another and they are all highly correlated to y, but that is redundant for our model.
3. What is a setting where you would use the L1-loss, and what is a setting where you would use L1-regularization? Answer: You use L_1-loss when there are outliers in your dataset and you use L_1-regularization for feature selections.
4. Among L0-regularization, L1-regularization, and L2-regularization: which yield convex objectives? Which yield unique solutions? Which yield sparse solutions? Answer: L_1, L_2 yield convex objectives, L_1, L_0 yield sparse solutions, L_2 yield unique 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? Answer: The greater λ is, the greater the sparsity of the solution and vice versa because λ dictates how much we penalize w for not reducing the number of features used. The greater λ is, the greater our training error is because we focus less on reducing the training error, but we also get a good approximation of test error because it tends to not overfit.
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. Answer: We can run our feature selection method to many bootstrap samples of our training data and select all features that are selected in at least a certain ratio (let's say for example 2/3) of our bootstrap samples.
7. Suppose a binary classification dataset has 3 features. If this dataset is "linearly separable", what does this precisely mean in three-dimensional space? Answer: It means that there exists a 2-dimensional plane partitioning the 3-dimensional feature space into two regions corresponding to the two binary labels.
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? Answer: Minimizing the number of classification errors means we use the 0-1 loss function, which is nonconvex and not smooth (so gradient descent cannot be used). If a perfect classifier exists, then we can find it with the perceptron algorithm, but in all other cases, minimizing the 0-1 loss function is difficult. We therefore seek an approximation to the 0-1 loss function which is easier to minimize. Logistic loss is particularly appealing because it is nondegenerate, convex, and differentiable.
9. What are "support vectors" and what's special about them? Answer: In a linear model of a binary classification problem, the support vectors are the examples closest to the hyperplane dividing the

feature space with respect to the L2 norm. For linearly separable data, SVM finds the linear model with largest distance to support vectors.

10. What is a disadvantage of using the perceptron algorithm to fit a linear classifier? Answer: If there is no way of knowing a priori whether the dataset is linearly separable, perceptron is not useful because in the case where the data is not linearly separable, training fails. Another flaw is that for certain datasets, absence of data means that perceptron may output a large range of different dividing planes and it cannot distinguish between any of these models.

11. Why we would use a multi-class SVM loss instead of using binary SVMs in a one-vs-all framework? Answer: n one-vs-all, all classifiers are fit independently, so the classifier for the label "c" is only trained to properly recognize whether the label "c" is correct or not. But we need the multi-classifier to be able to tell whether a label "c1" is more correct than another "c2".

12. How does the hyper-parameter σ affect the shape of the Gaussian RBFs bumps? How does it affect the fundamental tradeoff? Answer: σ controls width of Gaussian RBF bumps. Smaller σ means a more complex model, therefore lower training error and higher approximation error.