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

Yihao Zhang, #82626169, u6m1b
Hafsa Binte Zahid, #11668150, i6r0b

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).
   $f'(w) = 2\alpha w - \beta$, $f''(w) = 2\alpha$
   With $\alpha >= 0$, $f''(w) >= 0$ and it is a convex function.
2. $f(w) = -\log(\alpha w)$ with $\alpha > 0$ and $w > 0$ ("negative logarithm")
   $f'(w) = -\frac{1}{\alpha w} * \alpha = -\frac{1}{w}$, $f''(w) = \frac{1}{w^2}$, with $w > 0$, $f''(w) > 0$, and it is a convex function.
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. So $\|Xw - y\|_1$ is convex function since composition of a convex function and a linear function is also convex.
   norms are convex. So $\frac{\lambda}{2}\|w\|_1$ is convex. Both terms are convex, combined by an addition (linear). Thus, the function is also 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).
   Let $g(x) = log(1 + exp(x)), g'(x) = \frac{e^x}{1+e^x} = \frac{1}{e^{-x}+1}, g''(x) = \frac{e^{-x}}{(e^{-x}+1)^2} > 0$, Thus $g(x)$ is convex.
   $-y_i w^T x_i$ is linear function. $\log(1 + \exp(-y_i w^T x_i))$ is also convex since it is composition of $g(x)$ and $-y_i w^T x_i$.
   Sum of convex functions are convex. Thus, the function 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).
   Max is the L-inf norm, and norms are convex. The max of convex functions is convex. $|w^T x_i - y_i|$ is the norm of a linear function, so $\max\{0, |w^T x_i - y_i|\}$ is convex.
   Sum of convex functions is convex. So $\sum_{i=1}^{n} [\max\{0, |w^T x_i - y_i|\}$ is also convex.

$\|w\|_2^2$ is convex so $\frac{\lambda}{2}\|w\|_2^2$ is also function.
Thus, the function is convex since both terms are 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.
logReg Training error 0.000
logReg Validation error 0.084
nonZeros (features used): 101

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.
logRegL2 Training error 0.002
logRegL2 Validation error 0.074
nonZeros (features used): 101
36 gradient descent iterations.

```
class logRegL2(logReg):
    # L2 Regularized Logistic Regression
    def __init__(self, lammy=1.0, verbose=2, maxEvals=400):
        self.verbose = verbose
        self.L2_lambda = lammy
        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))) + \
            1/2 * self.L2_lambda * np.square(LA.norm(ord=None, x=w))

        # Calculate the gradient value
        res = - y / (1. + np.exp(yXw))
        g = X.T.dot(res) + self.L2_lambda * w

        return f, g
```

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.

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
nonZeros (features used): 71
78 iterations

```
class logRegL1(logReg):
    # L1 Regularized Logistic Regression
    def __init__(self, L1_lambda=1.0, verbose=2, maxEvals=400):
        self.verbose = verbose
        self.L1_lambda = L1_lambda
        self.maxEvals = maxEvals

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

        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.
Training error 0.000
Validation error 0.018
nonZeros (features selected): 24

```
        X_new = np.empty(shape=(len(X), len(selected_new)))

        counter = 0
        for col in selected_new:
```

```
            X_new[:, counter] = X[:, col]
            counter += 1

        self.w = np.zeros(len(selected_new))
        (self.w, f) = findMin.findMin(self.funObj, self.w,
                                      self.maxEvals, X_new, y, verbose=self.verbose)

        # add lambda * 0-norm
        if f + self.L0_lambda * (self.w != 0).sum() < minLoss:
            minLoss = f + self.L0_lambda * (self.w != 0).sum()
            bestFeature = i
```

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!

It is expected that all three regularization methods decrease the test error, because they all reduce overfitting; when we overfit less, we expect that the validation errors will decrease. In order of lowest validation error and number of features used, we get L0, L1 then L2 regularizations. L2 penalizes the largest values the most, but since the slope decreases as the weights get closer to 0, it doesn't manage to make them hit 0 exactly; the number of features used remains unchanged. L1 has fewer features than L2 because the constant gradient means that some feature weights are made 0. The L0 minimizes the number of features used; it only focuses on the features that make the largest contributions, so it overfits less to the training data than all the others and ends up with the lowest validation error. The L2 has a non-zero training error because adding the regularization term makes the model overfit less to the training data. We need a smaller value of λ in the L2 regularization, for example by setting it to 0.9, to get the same training error as the L1 and L0 regularization.

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_`.

scikit-learn l2 LogisticRegression Training error 0.002
scikit-learn l2 LogisticRegression Validation error 0.074
scikit-learn l2 LogisticRegression has 101 of nonZeros
scikit-learn l1 LogisticRegression Training error 0.0
scikit-learn l1 LogisticRegression Validation error 0.052
scikit-learn l1 LogisticRegression has 71 of nonZeros
Our results are the same with skilit-learn's LogisticRegression.

```
    regularization = ['l2', 'l1']
    for r in regularization:
        model = LogisticRegression(penalty=r, C=1.0, fit_intercept=False)
```

```
model.fit(XBin, yBin)

print("scikit-learn {} LogisticRegression Training error {}"
      .format(r, utils.classification_error(model.predict(XBin), yBin)))
print("scikit-learn {} LogisticRegression Validation error {}"
      .format(r, utils.classification_error(model.predict(XBinValid), yBinValid)))
print("scikit-learn {} LogisticRegression has {} of nonZeros"
      .format(r, ((model.coef_[0] != 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.

$$\begin{aligned} f(w) &= \frac{1}{2}(w-2)^2 + \frac{1}{2}(0-1)^2 + \lambda\sqrt{|w|} \\ &= \frac{1}{2}[(w-2)^2 + 1] + \lambda\sqrt{|w|} \end{aligned}$$

2. If $\lambda = 0$, what is the solution, i.e. $\arg\min_w f(w)$?
   With $\lambda = 0$, $f(w) = \frac{1}{2}[(w-2)^2 + 1]$.
   $\arg\min_w f(w) = 2$

3. If $\lambda \to \infty$, what is the solution, i.e., $\arg\min_w f(w)$?
   With $\lambda \to \infty$, $\lambda|w|$ becomes dominant term. $\arg\min_w f(w) = 0$ to set the term to minimum of zero.

4. Plot $f(w)$ when $\lambda = 1$. What is $\arg\min_w f(w)$ when $\lambda = 1$? Answer to one decimal place if appropriate.
   argmin of f(w) is $w = 1.6049999999935682$ with $\lambda = 1$

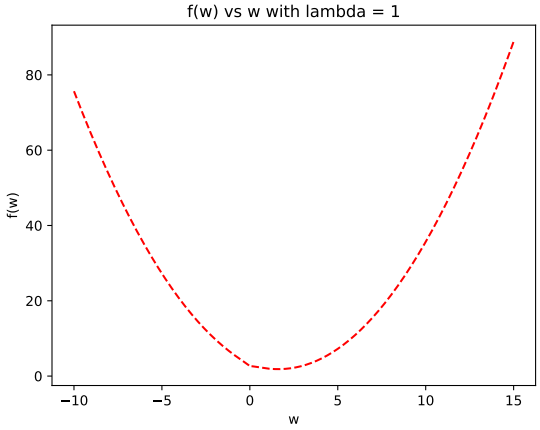

Figure 1: Loss vs w with $\lambda = 1$

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

```
x = np.linspace(-10,15,20000)
    y = []
    for w in x:
        b = 1/2*((w-2)**2+1)+10*np.sqrt(np.linalg.norm(w))
        y.append(b)
    b_min = x[np.argmin(y)]
    print(b_min)
    plt.plot(x,y)
    plt.show()
```

argmin of f(w) is $w = 0.00050002500125$ with $\lambda = 10$, which is the closest value to 0 that exists within the values in our range. If our x array contained the value 0 as a a data point, the minimum point would be at w=0

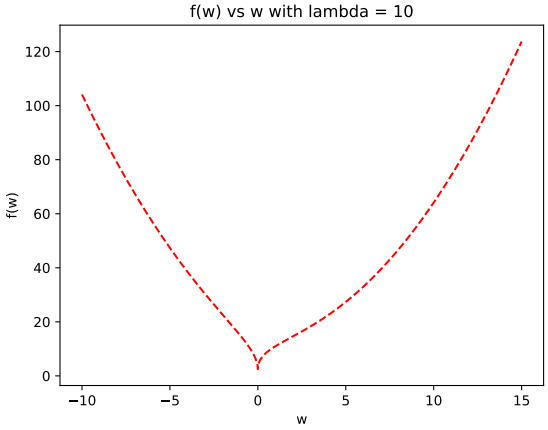

Figure 2: Loss vs w with $\lambda = 10$

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 than L2 regularization for feature selection. $L\frac{1}{2}$ adds $\lambda\sqrt{|w|}$ which is a concave function to least squares. Thus, it encourages even more zeros in w than L1 whereas L2 moves w of irrelevant features close to zero but not exactly zero.
   To verify:
   Same equation with L1: argmin of f(w) is $w = 0.00050002500125$ with $\lambda = 10$ which gives the same result as $L\frac{1}{2}$.
   Same equation with L2: argmin of f(w) is w = 0.09499999999440512 with $\lambda = 10$.

7. Is least squares with $L\frac{1}{2}$ regularization a convex optimization problem? Briefly justify your answer.
   It is not a convex optimization problem since $L\frac{1}{2}$ has a nonconvex penalty. As showed in 2.6.4, the plot givs a non-convex property near 0.

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 class 1: $w^T x = 2 * 1 + (-1) * 1 = 1$
For class 2: $w^T x = 2 * 1 + (-2) * 1 = 0$
For class 3: $w^T x = 3 * 1 + (-1) * 1 = 2$
Thus, the model will assign test example to 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.
logLinearClassifier Training error 0.084
logLinearClassifier Validation error 0.070

```
class logLinearClassifier:
    def __init__(self, maxEvals=500, verbose=0):
        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):
            ytmp = y.copy().astype(float)
            ytmp[y==i] = 1
            ytmp[y!=i] = -1
            (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

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

Taking the partial derivative of W_{cj} over a sum turns all values in the sum W_y to 0, except when $y = c$
Let $g(w) = \log\sum_{c'=1}^{k}\exp(w_{c'}^T x_i)$
Then, $\frac{\partial g}{\partial W_{cj}} = x_{ij} * \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k}\exp(w_{c'}^T x_i)} = x_{ij} * p(y_i = c \mid W, x_i)$
Let $h(w) = -w_{y_i}^T x_i$
Then, $\frac{\partial h}{\partial W_{cj}} = -x_{ij} * \delta(y_i - c) = -x_{ij} * I(y_i = c)$
We can write $f(w) = \sum_{i=1}^{n}(h(w) + g(w))$
Then, using the definitions above, $\frac{\partial f}{\partial W_{cj}} = \sum_{i=1}^{n}(\frac{\partial h}{\partial W_{cj}} + \frac{\partial g}{\partial W_{cj}}) = \sum_{i=1}^{n} x_{ij}[p(y_i = c \mid W, x_i) - I(y_i = c)]$

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.
Training error 0.000
Validation error 0.008

```
class softmaxClassifier():
    def __init__(self, maxEvals):
        self.maxEvals = maxEvals
        pass

    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))
        W_vector = self.W.flatten()
        (self.W, f) = findMin.findMin(self.softmaxFun, W_vector,
                                      self.maxEvals, X, y, verbose=0)

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

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

    def softmaxFun(self, W, X, y):
        n, d = X.shape
        k = self.n_classes
```

```
    W = np.reshape(W, (k, d))
    XWT = X.dot(W.T)

    # softmax loss
    f = 0
    for i in range(n):
        sum_exp = 0
        for c in range(k):
            sum_exp += np.exp(XWT[i, c])
        f += -1 * XWT[i, y[i]] + np.log(sum_exp)

    # gradient
    I = np.zeros((n,k))
    p = np.zeros((n,k))

    for i in range(n):
        for c in range(k):
            sum_exp = 0
            for c_prime in range(k):
                sum_exp += np.exp(XWT[i, c_prime])
            p[i, c] = np.exp(XWT[i, c]) / sum_exp
            if y[i] == c:
                I[i, c] = 1
    res = p - I
    g = X.T.dot(res).T # X.T.dot(res) refers to Wc,j (d*k) our W is (k*d)

    return f, g.flatten()
```

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).
scikit-learn LogisticRegression One-vs-all:
Training error 0.084
Validation error 0.070
scikit-learn LogisticRegression softmax:
Training error 0.000
Validation error 0.008
Our results are the same as scikit-learn's LogisticRegression.

```
model = LogisticRegression(C=10000,fit_intercept=False)
model.fit(XMulti, yMulti)
print("scikit-learn LogisticRegression One-vs-all:")
print("Training error %.3f" %
      utils.classification_error(model.predict(XMulti), yMulti))
print("Validation error %.3f" %
      utils.classification_error(model.predict(XMultiValid), yMultiValid))
```

```
model = LogisticRegression(C=10000,fit_intercept=False,
                           solver='lbfgs', multi_class='multinomial')
model.fit(XMulti, yMulti)
print("scikit-learn LogisticRegression softmax:")
print("Training error %.3f" %
      utils.classification_error(model.predict(XMulti), yMulti))
print("Validation error %.3f" %
      utils.classification_error(model.predict(XMultiValid), yMultiValid))
```

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?
   RBF transform takes an $nx1$ vector X and outputs an nxn matrix Z. This takes $O(n^2)$ time. Then for each iteration, we need to compute f, which has two summations; one goes 1 to n, the other goes 1 to k. W is a kxn matrix, so $\sum_{c'=1}^{k} \exp(w_{c'}^T z_i)$ takes $O(kn)$. Computing f takes $O(kn + kn^2) = O(kn^2)$. We also need to compute g for each iteration, which also has a $\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)$ summation within a sum going from 1 to n; so, it has the same order as f, $O(kn^2)$. Since Z is formed once, and f and g are formed on every iteration, the total time is $O(n^2 + kn^2T)$

2. What is the cost of classifying the t test examples?
   We get an input $\tilde{X}$ which is $tx1$, and form a txn matrix $\tilde{Z}$, which takes $O(tn)$ time. V is an nxk vector, so $\tilde{Z}$*V takes $O(nkt)$ time. So the total time taken is $O(tn + nkt)$

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?
   Relevance is relative to what other information is available in the dataset. In the Tacos on Tuesdays example given in class, Tuesday was identified as a relevant feature when given a certain set of data where tacos were not eaten on other days, even though the day of the week is irrelevant. So, whether

or not we can identify relevant features depends on what kinds of information is contained in the dataset.

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?
   In the Mentos and Coke example, the mutual information is only relevant when they are each above a certain threshold; they are only relevant when taken together (eg. 0.5 mentos + 0.5 coke), but they are irrelevant when the other is not present (eg. 1 mentos + 0 coke or 0 mentos + 1 coke). If we have a threshold of mutual information = 1, all of "mentos = 1, coke = 0", "mentos = 0, coke = 1" and "mentos = 0.5, coke = 0.5" will be considered relevant, but in reality it's only relevant for "mentos = 0.5, coke = 0.5".

3. What is a setting where you would use the L1-loss, and what is a setting where you would use L1-regularization?
   If we want to remove outliers (irrelevant examples), we would use L1-loss. If we want to remove irrelevant features, 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?
   L1 and L2 yield convex objectives. L2 gives unique solution. L0 and L1 yield 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?
   With increasing λ, the sparsity increases. Increasing λ would increase the training error as it makes w smaller, and at the same time, it decreases the approximation error.

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.
   Assuming we have a representative dataset, we can ensemble the current method with regression weight approach which take all feature j into account when w_j is greater than a threshold. It will reduce false negatives.

7. Suppose a binary classification dataset has 3 features. If this dataset is "linearly separable", what does this precisely mean in three-dimensional space?
   Classes can be seperated by a 2D 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 it's noon-smooth and non-convex, which make finding w harder. Even with convex approximation, it has the issue of degenerate solution and logistic loss doesn't.

9. What are "support vectors" and what's special about them?
   Support vectors are the points closest to the decision boundary between separable features. They are used to maximize the margins, which is to get the largest distance between the boundary and the closest examples. This helps make the boundary more robust

10. What is a disadvantage of using the perceptron algorithm to fit a linear classifier?
    Perceptron algrithm is difficult to classify data which are not linearly separable.

11. Why we would use a multi-class SVM loss instead of using binary SVMs in a one-vs-all framework?
    One-vs-all classifier can hurt performance because classifiers are fit independently. So there is no

attempt to calibrate the columns of the matrix W.

12. How does the hyper-parameter σ affect the shape of the Gaussian RBFs bumps? How does it affect the fundamental tradeoff?
σ (or σ^2) is the hypermeter controlling the width of bumps. With smaller σ, the bumps are narrower. With smaller σ, the training error will decrease but approximation error will increase.

Appendices

A linear_model.py

```
import numpy as np
from numpy.linalg import solve
import findMin
from scipy.optimize import approx_fprime
import utils
from numpy import linalg as LA


class logReg:
    # Logistic Regression
    def __init__(self, verbose=0, maxEvals=100):
        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)))
        # 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.findMin(self.funObj, self.w,
                                      self.maxEvals, X, y, verbose=self.verbose)
    def predict(self, X):
        return np.sign(X@self.w)


class logRegL2(logReg):
    # L2 Regularized Logistic Regression
    def __init__(self, lammy=1.0, verbose=2, maxEvals=400):
        self.verbose = verbose
        self.L2_lambda = lammy
        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))) + \
```

```
            1/2 * self.L2_lambda * np.square(LA.norm(ord=None, x=w))

        # Calculate the gradient value
        res = - y / (1. + np.exp(yXw))
        g = X.T.dot(res) + self.L2_lambda * w

        return f, g


class logRegL1(logReg):
    # L1 Regularized Logistic Regression
    def __init__(self, L1_lambda=1.0, verbose=2, maxEvals=400):
        self.verbose = verbose
        self.L1_lambda = L1_lambda
        self.maxEvals = maxEvals

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

        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)

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

                X_new = np.empty(shape=(len(X), len(selected_new)))

                counter = 0
                for col in selected_new:
                    X_new[:, counter] = X[:, col]
                    counter += 1

                self.w = np.zeros(len(selected_new))
                (self.w, f) = findMin.findMin(self.funObj, self.w,
                                              self.maxEvals, X_new, y, verbose=self.verbose)

                # add lambda * 0-norm
                if f + self.L0_lambda * (self.w != 0).sum() < minLoss:
                    minLoss = f + self.L0_lambda * (self.w != 0).sum()
                    bestFeature = i

            selected.add(bestFeature)

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


class leastSquaresClassifier:
    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] = np.linalg.solve(X.T@X+0.0001*np.eye(d), X.T@ytmp)

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


class logLinearClassifier:
    def __init__(self, maxEvals=500, verbose=0):
        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):
            ytmp = y.copy().astype(float)
            ytmp[y==i] = 1
            ytmp[y!=i] = -1
            (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)


class softmaxClassifier():
    def __init__(self, maxEvals):
        self.maxEvals = maxEvals
        pass

    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))
        W_vector = self.W.flatten()
        (self.W, f) = findMin.findMin(self.softmaxFun, W_vector,
                                      self.maxEvals, X, y, verbose=0)

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

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

    def softmaxFun(self, W, X, y):
        n, d = X.shape
        k = self.n_classes

        W = np.reshape(W, (k, d))
        XWT = X.dot(W.T)

        # softmax loss
        f = 0
        for i in range(n):
            sum_exp = 0
            for c in range(k):
                sum_exp += np.exp(XWT[i, c])
```

```
            f += -1 * XWT[i, y[i]] + np.log(sum_exp)

        # gradient
        I = np.zeros((n,k))
        p = np.zeros((n,k))

        for i in range(n):
            for c in range(k):
                sum_exp = 0
                for c_prime in range(k):
                    sum_exp += np.exp(XWT[i, c_prime])
                p[i, c] = np.exp(XWT[i, c]) / sum_exp
                if y[i] == c:
                    I[i, c] = 1
        res = p - I
        g = X.T.dot(res).T # X.T.dot(res) refers to Wc,j (d*k) our W is (k*d)

        return f, g.flatten()
```

B main.py

```
import argparse
import numpy as np
import os
import utils
import linear_model
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('-q','--question', required = True)
    io_args = parser.parse_args()
    question = io_args.question

    if question == "1":
        W = np.zeros((3, 2))
        W[0,0] = 2
        W[0,1] = -1
        W[1,0] = 2
        W[1,1] = -2
        W[2,0] = 3
        W[2,1] = -1

        print(W)
        print("----------")
        print(W.T)
        print("----------")
        print(np.reshape(W, (2,3)))

    if question == "2":
        data = utils.load_dataset("logisticData")
        XBin, yBin = data['X'], data['y']
        XBinValid, yBinValid = data['Xvalid'], data['yvalid']
```

```
    model = linear_model.logReg(maxEvals=400)
    model.fit(XBin,yBin)

    print("\nlogReg Training error %.3f" % utils.classification_error(model.predict(XBin), y
    print("logReg Validation error %.3f" % utils.classification_error(model.predict(XBinVali
    print("# nonZeros: %d" % (model.w != 0).sum())

elif question == "2.1":
    data = utils.load_dataset("logisticData")
    XBin, yBin = data['X'], data['y']
    XBinValid, yBinValid = data['Xvalid'], data['yvalid']

    model = linear_model.logRegL2(lammy=0.9, maxEvals=400)
    model.fit(XBin,yBin)

    print("\nlogRegL2 Training error %.3f" % utils.classification_error(model.predict(XBin),
    print("logRegL2 Validation error %.3f" % utils.classification_error(model.predict(XBinVa
    print("\# nonZeros (features used): %d" % (model.w != 0).sum())

elif question == "2.2":
    data = utils.load_dataset("logisticData")
    XBin, yBin = data['X'], data['y']
    XBinValid, yBinValid = data['Xvalid'], data['yvalid']

    model = linear_model.logRegL1(L1_lambda=1.0, maxEvals=400)
    model.fit(XBin,yBin)

    print("\nlogRegL1 Training error %.3f" % utils.classification_error(model.predict(XBin),
    print("logRegL1 Validation error %.3f" % utils.classification_error(model.predict(XBinVa
    print("\# nonZeros (features used): %d" % (model.w != 0).sum())

elif question == "2.3":
    data = utils.load_dataset("logisticData")
    XBin, yBin = data['X'], data['y']
    XBinValid, yBinValid = data['Xvalid'], data['yvalid']

    model = linear_model.logRegL0(L0_lambda=1.0, maxEvals=400)
    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), yBi
    print("# nonZeros: %d" % (model.w != 0).sum())

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

    # TODO
    regularization = ['l2', 'l1']
    for r in regularization:
        model = LogisticRegression(penalty=r, C=1.0, fit_intercept=False)
        model.fit(XBin, yBin)

        print("scikit-learn {} LogisticRegression Training error {}"
              .format(r, utils.classification_error(model.predict(XBin), yBin)))
```

```
        print("scikit-learn {} LogisticRegression Validation error {}"
              .format(r, utils.classification_error(model.predict(XBinValid), yBinValid)))
        print("scikit-learn {} LogisticRegression has {} of nonZeros"
              .format(r, ((model.coef_[0] != 0).sum())))

elif question == "2.6":
    # w = np.arange(-10, 15, 0.0001)
    w = np.linspace(-1, 1.5, 100000)
    # lam = 1
    # f_1 = 1/2 * np.power(w-2, 2) + 1/2 + lam * np.sqrt(abs(w))
    # plt.plot(w, f_1, 'r--')
    # plt.xlabel("w")
    # plt.ylabel("f(w)")
    # plt.title("f(w) vs w with lambda = 1")
    # fname = os.path.join("..", "figs", "q2_6_lambda_1.pdf")
    # plt.savefig(fname)
    # print("argmin of f(w) is w = {} with lambda = {}".format(w[np.argmin(f_1)], lam))
    lam = 10
    f_10 = 1/2 * np.power(w-2, 2) + 1/2 + lam * np.sqrt(abs(w))
    plt.plot(w, f_10, 'r--')
    plt.xlabel("w")
    plt.ylabel("f(w)")
    plt.title("f(w) vs w with lambda = 10")
    fname = os.path.join("..", "figs", "q2_6_lambda_10.pdf")
    plt.savefig(fname)
    print("argmin of f(w) is w = {} with lambda = {}".format(w[np.argmin(f_10)], lam))


elif question == "3":
    data = utils.load_dataset("multiData")
    XMulti, yMulti = data['X'], data['y']
    XMultiValid, yMultiValid = data['Xvalid'], data['yvalid']

    model = linear_model.leastSquaresClassifier()
    model.fit(XMulti, yMulti)

    print("leastSquaresClassifier Training error %.3f" % utils.classification_error(model.pr
    print("leastSquaresClassifier Validation error %.3f" % utils.classification_error(model.

    print(np.unique(model.predict(XMulti)))


elif question == "3.2":
    data = utils.load_dataset("multiData")
    XMulti, yMulti = data['X'], data['y']
    XMultiValid, yMultiValid = data['Xvalid'], data['yvalid']

    model = linear_model.logLinearClassifier(maxEvals=500, verbose=0)
    model.fit(XMulti, yMulti)

    print("logLinearClassifier Training error %.3f" % utils.classification_error(model.predi
    print("logLinearClassifier Validation error %.3f" % utils.classification_error(model.pre

elif question == "3.4":
    data = utils.load_dataset("multiData")
    XMulti, yMulti = data['X'], data['y']
```

```
    XMultiValid, yMultiValid = data['Xvalid'], data['yvalid']

    model = linear_model.softmaxClassifier(maxEvals=500)
    model.fit(XMulti, yMulti)

    print("Training error %.3f" % utils.classification_error(model.predict(XMulti), yMulti))
    print("Validation error %.3f" % utils.classification_error(model.predict(XMultiValid), y

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=10000,fit_intercept=False)
    model.fit(XMulti, yMulti)
    print("scikit-learn LogisticRegression One-vs-all:")
    print("Training error %.3f" %
          utils.classification_error(model.predict(XMulti), yMulti))
    print("Validation error %.3f" %
          utils.classification_error(model.predict(XMultiValid), yMultiValid))
    model = LogisticRegression(C=10000,fit_intercept=False,
                               solver='lbfgs', multi_class='multinomial')
    model.fit(XMulti, yMulti)
    print("scikit-learn LogisticRegression softmax:")
    print("Training error %.3f" %
          utils.classification_error(model.predict(XMulti), yMulti))
    print("Validation error %.3f" %
          utils.classification_error(model.predict(XMultiValid), yMultiValid))
```