CPSC 340 Assignment 4

Important: Submission Format [5 points]

Please make sure to follow the submission instructions posted on the course website. We will deduct marks if the submission format is incorrect, or if you're not using LaTeX and your submission is *at all* difficult to read – at least these 5 points, more for egregious issues.

1 Convex Functions [15 points]

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: We can prove by showing second derivative is always greater or equal to 0.
   $f(w) = \alpha w^2 - \beta w + \gamma$
   $f'(w) = 2\alpha w - \beta$
   $f''(w) = 2\alpha >= 0$, as $\alpha >= 0$ (shown)

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

   Answer: We can prove by showing the second derivative is always greater than or equal to 0.
   $f(w) = -\log(\alpha w)$
   $f'(w) = -\frac{\alpha}{\alpha w} = -\frac{1}{w}$
   $f''(w) = \frac{1}{w^2}$ (shown)

3. $f(w) = \|Xw - y\|_2 + \frac{\lambda}{2}\|w\|_1$ with $w \in \mathbb{R}^d, \lambda \geq 0$ (L1-regularized linear regression).

   Answer: Norms are always convex, therefore $\|Xw-y\|_2$ and $\|w\|_1$ are convex. A non-negative constant, $\lambda/2$ multiplied by a convex function is always convex. Therefore, the two halves of the addition are convex. Next, the sum of convex functions are always functions. Therefore the entire function 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: Let $z = -y_i w^T x_i$, such that $f(z) = \log(1 + \exp(z))$. We show that $f(z)$ is convex by finding the second derivative:

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

   (simplified using the provided expression in the question).

   $$f''(z) = \frac{\exp(z)}{(1 + \exp(-z))^2} \geq 0$$

   regardless of the z-value, since both the numerator and denominator are always greater than or equal to 0.

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: First term of the summation: 0 and $|w^T x_i - y_i| - \epsilon$ are both convex functions so the max function $max(0, |w^T x_i - y_i| - \epsilon)$ is also convex. Since the summation is a summation of the convex max functions, $\Sigma_{i=1}^{n} max(0, |w^T x_i - y_i| - \epsilon)$, the summation is also convex.

   Second term of the summation:Norms are always convex so $||w||_2^2$ is convex. A non-negative constant, $\lambda/2$ multiplied by a convex function is always convex.

   Since both the left and right functions of $f(w)$ is a sum of two convex functions, $f(w)$ 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, it's easier 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 at first seem non-convex since it contains $\log(z)$ and log is concave, but note that $\log(\exp(z)) = z$ 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 [30 points]

If you run `python main.py 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 slightly, depending on your software versions, the exact order you do floating-point operations in, and so on.

2.1 L2-Regularization [5 points]

In `linear_models.py`, you will find a class named `LinearClassifier` that defines the fitting and prediction behaviour of a logistic regression classifier. As with ordinary least squares linear regression, the particular choice of a function object (`fun_obj`) and an optimizer (`optimizer`) will determine the properties of your output model. Your task is to implement a logistic regression classifier that uses L2-regularization on its weights. Go to `fun_obj.py` and complete the `LogisticRegressionLossL2` class. This class' constructor takes an input parameter λ, the L2 regularization weight. Specifically, while `LogisticRegressionLoss` computes

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

your new class `LogisticRegressionLossL2` should compute

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

and its gradient. Submit your function object code. Using this new code with $\lambda = 1$, report how the following quantities change: (1) the training (classification) error, (2) the validation (classification) error, (3) the number of features used, and (4) the number of gradient descent iterations.

Answer:
Using LogisticRegression:
Linear Training error: 0.000
Linear Validation error: 0.082
Number of Features: 101
Number of Gradient Descent Iterations: 89

Using newly implemented LogisticRegressionL2:
Linear Training error: 0.002
Linear Validation error: 0.074
Number of Features: 101
Number of Gradient Descent Iterations: 30

As visible above, the training error went up by 0.002, but the validation error went down by 0.008. The number of features remains the same at 101, but the number of gradient descent iterations drop significantly by 59; from 89 to 30.

```python
# linear_models.py
class LogisticRegressionLossL2(LogisticRegressionLoss):
    def __init__(self, lammy):
        super().__init__()
        self.lammy = lammy

    def evaluate(self, w, X, y):
        # help avoid mistakes by potentially reshaping our arguments
        w = ensure_1d(w)
        y = ensure_1d(y)

        reg_term = self.lammy/2 * (w.T @ w)

        Xw = X @ w
        yXw = y * Xw  # element-wise multiply; the y_i are in {-1, 1}

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

        # Calculate the gradient value
        s = -y / (1 + np.exp(yXw))
        g = X.T @ s + (self.lammy * w)

        return f, g

```

2.2 L1-Regularization and Regularization Path [5 points]

L1-regularized logistic regression classifier has the following objective function:

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

Because the L1 norm isn't differentiable when any elements of w are 0 – and that's *exactly what we want to get* – standard gradient descent isn't going to work well on this objective. There is, though, a similar approach called *proximal gradient descent* that does work here.

This is implemented for you in the `GradientDescentLineSearchProxL1` class inside `optimizers.py`. Note that to use it, you *don't include the L1 penalty in your loss function object*; the optimizer handles that itself.

Write and submit code to instantiate `LinearClassifier` with the correct function object and optimizer for L1-regularization. Using this linear model, obtain solutions for L1-regularized logistic regression with $\lambda = 0.01$, $\lambda = 0.1$, $\lambda = 1$, $\lambda = 10$. Report the following quantities per each value of λ: (1) the training error, (2) the validation error, (3) the number of features used, and (4) the number of gradient descent iterations.

Answer:
Using $\lambda = 0.01$:
Linear Training error: 0.000
Linear Validation error: 0.072
Number of Features: 89
Number of Gradient Descent Iterations: 158

Using $\lambda = 0.1$:
Linear Training error: 0.000
Linear Validation error: 0.060
Number of Features: 81
Number of Gradient Descent Iterations: 236

Using $\lambda = 1$:
Linear Training error: 0.000
Linear Validation error: 0.052
Number of Features: 71
Number of Gradient Descent Iterations: 107

Using $\lambda = 10$:
Linear Training error: 0.050
Linear Validation error: 0.090
Number of Features: 29
Number of Gradient Descent Iterations: 14

```python
# main.py
@handle("2.2")
def q2_2():
    data = load_dataset("logisticData")
    X, y = data["X"], data["y"]
    X_valid, y_valid = data["Xvalid"], data["yvalid"]

    lamArray = [0.01,0.1, 1, 10]

    for lam in lamArray:
        fun_obj = LogisticRegressionLoss()
        optimizer = GradientDescentLineSearchProxL1(max_evals=400, verbose=False,
         ↪  lammy=lam)
        model = linear_models.LinearClassifier(fun_obj, optimizer)
        model.fit(X, y)

        train_err = classification_error(model.predict(X), y)
        print("For lambda value of " + str(lam))
        print(f"Linear Training error: {train_err:.3f}")

        val_err = classification_error(model.predict(X_valid), y_valid)
        print(f"Linear Validation error: {val_err:.3f}")

        print(f"Number of Features: {np.sum(model.w != 0)}")
        print(f"Number of Gradient Descent Iterations: {optimizer.num_evals}")
        print("")

```

2.3 L0 Regularization [8 points]

The class `LogisticRegressionLossL0` in `fun_obj.py` 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 class `LinearClassifierForwardSel` in `linear_models.py` will use a loss function object and an optimizer to perform a forward selection to approximate the best feature set. The `for` loop in its `fit()` method is missing the part where we fit the model using the subset `selected_new`, then compute the score and updates the `min_loss` and `best_feature`. 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 variables `min_loss` and `best_feature`, as well as `self.total_evals`. Hand in your updated code. Using this new code with $\lambda = 1$, report the training error, validation error, number of features selected, and total optimization steps.

Note that the code differs slightly from what we discussed in class, since we're hard-coding that we include the first (bias) variable. Also, note that for this particular case using the L0-norm with $\lambda = 1$ is using 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.

Answer:
Linear training 0-1 error: 0.000
Linear validation 0-1 error: 0.018
Number of features: 24
Total function evaluations: 2,353

```python
# linear_model.py
class LinearClassifierForwardSel(LinearClassifier):
    def fit(self, X, y):
        n, d = X.shape

        # Maintain the set of selected indices, as a boolean mask array.
        # We assume that feature 0 is a bias feature, and include it by default.
        selected = np.zeros(d, dtype=bool)
        selected[0] = True
        min_loss = np.inf
        self.total_evals = 0

        # We will hill-climb until a local discrete minimum is found.
        while not np.all(selected):
            old_loss = min_loss
            print(f"Epoch {selected.sum():>3}:", end=" ")

            for j in range(d):
                if selected[j]:
                    continue

                selected_with_j = selected.copy()
                selected_with_j[j] = True

                """YOUR CODE HERE FOR Q2.3"""
                # TODO: Fit the model with 'j' added to the features,
                # then compute the loss and update the min_loss/best_feature.
                # Also update self.total_evals.
                w = np.zeros(np.sum(selected_with_j))
                w, fs, gs, ws = self.optimize(w, X[:, selected_with_j], y)
                loss, _ = self.global_loss_fn.evaluate(w, X[:, selected_with_j], y)
                if loss < min_loss:
                    min_loss = loss
                    best_feature = j
                self.total_evals += 1

            if min_loss < old_loss:  # something in the loop helped our model
                selected[best_feature] = True
                print(f"adding feature {best_feature:>3} - loss {min_loss:>7.3f}")
            else:
                print("nothing helped to add; done.")
                break
        else:  # triggers if we didn't break out of the loop
            print("wow, we selected everything")

        w_init = np.zeros(selected.sum())
        w_on_sub, *_ = self.optimize(w_init, X[:, selected], y)
        self.total_evals += self.optimizer.num_evals

        self.w = np.zeros(d)
        self.w[selected] = w_on_sub
```

2.4 Discussion [4 points]

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:
Different norms were tested for logistical regression, and the resulting ranking of the best to worst validation error is: L0, L1, L2, No regularization. The trend here that is improving the validation error is the degree of sparsity. L0 essentially optimizes the features that make the best predictions, whereas L2 and no regularization do not introduce any sparsity. Having a model that helps optimize features reduces model complexity (i.e. less features used in L0 and L1) and therefore improves validation error. However the issue with L0 is that it is computationally expensive and takes a lot of time as it needs to test each possible combination of features.

2.5 L$\frac{1}{2}$ regularization [8 points]

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 the very special case of $n = 2$, where our 2 data points are $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (0, 1)$.

1. Plug in the dataset values and write the loss in a simplified form, without a $\sum$.

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

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

4. Plot $f(w)$ when $\lambda = 1$. What is $\arg\min_w f(w)$ when $\lambda = 1$? Answer to one decimal place if appropriate. (For the plotting questions, you can use `matplotlib` or any graphing software, such as `https://www.desmos.com`.)

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

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: As lambda increases in value L$\frac{1}{2}$ acts like L1 as w decreases in size with higher lambda, introducing sparsity and feature selection.

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

   Answer: $\lambda\sqrt{|w|}$ is non-convex, so even though the first part of the summation is convex, the resulting function $f(w)$ is not convex. Therefore, solving this is not a convex optimization problem.

3 Multi-Class Logistic Regression [32 points]

If you run `python main.py 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 [4 points]

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:

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

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

$$W = \begin{bmatrix} +4 & -1 \\ +2 & -2 \\ +3 & -1 \end{bmatrix}$$

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

Answer: If class 1, $z_1 = (4)(1) + (-1)(1) = 3$.
If class 2, $z_2 = (2)(1) + (-2)(1) = 0$.
If class 3, $z_3 = (3)(1) + (-1)(1) = 2$.

Or, in matrix notation,

$$W\mathbf{\tilde{x}^T} = \begin{bmatrix} 4 & -1 \\ 2 & -2 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 2 \end{bmatrix}.$$

The model will predict class 1 as it has the highest z-value.

3.2 One-vs-all Logistic Regression [7 points]

Using the squared error on this problem hurts performance because it has "bad errors" (the model gets penalized if it classifies examples "too correctly"). In `linear_models.py`, complete the class named `LinearClassifierOneVsAll` 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:
LinearClassifierOneVsAll training 0-1 error: 0.084
LinearClassifierOneVsAll validation 0-1 error: 0.070

```python
class LinearClassifierOneVsAll(LinearClassifier):
    """
    Uses a function object and an optimizer.
    """

    def fit(self, X, y):
        n, d = X.shape
        y_classes = np.unique(y)
        k = len(y_classes)
        assert set(y_classes) == set(range(k))  # check labels are {0, 1, ..., k-1}

        # quick check that loss_fn is implemented correctly
        self.loss_fn.check_correctness(np.zeros(d), X, (y == 1).astype(np.float32))

        # Initial guesses for weights
        W = np.zeros([k, d])

        """YOUR CODE HERE FOR Q3.2"""
        # NOTE: make sure that you use {-1, 1} labels y for logistic regression,
        #       not {0, 1} or anything else.
        for c in y_classes:
            y_c = np.where(c == y, 1, -1)
            w_c = np.zeros(d)
            w_c, _, _, _ = self.optimize(w_c, X, y_c)
            W[c, :] = w_c

        self.W = W

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

3.3 Softmax Classifier Gradient [7 points]

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:

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

where...

- $\mathbb{1}(y_i = c)$ 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:

$$f(W) = f_{left} + f_{right}$$

$$f_{left} = \sum_{i=1}^{n} -w_{y_i}^T x_i$$

$$f_{left} = \sum_{i=1}^{n}\sum_{j=1}^{d} -w_{cj} x_{ij}$$

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

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

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

$$\frac{\partial f_{right}}{\partial W_{cj}} = \sum_{i=1}^{n} x_{ij} p(y_i = c | W, x_i)$$

$$\frac{\partial f}{\partial W_{cj}} = \frac{\partial f_{left}}{\partial W_{cj}} + \frac{\partial f_{right}}{\partial W_{cj}}$$

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

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

3.4 Softmax Classifier Implementation [8 points]

Inside `linear_models.py`, you will find the class `MulticlassLinearClassifier`, which fits W using the softmax loss from the previous section instead of fitting k independent classifiers. As with other linear models, you must implement a function object class in `fun_obj.py`. Find the class named `SoftmaxLoss`. Complete these classes and their methods. Submit your code and report the validation error.

Hint: You may want to use `check_correctness()` 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 our `optimizers.py`) 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 skeleton code of `SoftmaxLoss` already has lines reshaping the input vector w into a $k \times d$ matrix using `np.reshape`. You can then compute the gradient using sane, readable code with the W matrix inside `evaluate()`. You'll end up with a gradient that's also a matrix: one partial derivative per element of W. Right at the end of `evaluate()`, you can flatten this gradient matrix into a vector using `g.reshape(-1)`. If you do this, the optimizer will be sending in a vector of parameters to `SoftmaxLoss`, and receiving a gradient vector back out, which is the interface it wants – and your `SoftmaxLoss` code will be much more readable, too. You may need to do a bit more reshaping elsewhere, but this is the key piece.

Hint: A naïve implementation of `SoftmaxLoss.evaluate()` might involve many for-loops, which is fine as long as the function and gradient calculations are correct. However, this method might take a very long time! This speed bottleneck is one of Python's shortcomings, which can be addressed by employing pre-computing and lots of vectorized operations. However, it can be difficult to convert your written solutions of f and g into vectorized forms, so you should prioritize getting the implementation to work correctly first. One reasonable path is to first make a correct function and gradient implementation with lots of loops, then (if you want) pulling bits out of the loops into meaningful variables, and then thinking about how you can compute each of the variables in a vectorized way. Our solution code doesn't contain any loops, but the solution code for previous instances of the course actually did; it's totally okay for this course to not be allergic to Python `for` loops the way Danica is.[1]

[1]Reading the old solution with loops *probably* isn't why I was sick the last week. . . .

Answer:
SoftmaxLoss training 0-1 error: 0.000
SoftmaxLoss validation 0-1 error: 0.008

```python
# fun_obj.py
class SoftmaxLoss(FunObj):
    def evaluate(self, w, X, y):
        w = ensure_1d(w)
        y = ensure_1d(y)

        n, d = X.shape
        k = len(np.unique(y))

        """YOUR CODE HERE FOR Q3.4"""
        # Hint: you may want to use NumPy's reshape() or flatten()
        # to be consistent with our matrix notation.
        W = np.reshape(w, (k, d))
        XWT = X @ W.T
        exp_XWT = np.exp(XWT)
        sum_exp_XWT = np.sum(exp_XWT, axis=1)
        log_sum_exp_XWT = np.log(sum_exp_XWT)
        f_right_term = np.sum(log_sum_exp_XWT)
        f_left_term = 0

        for i in range(n):
            f_left_term -= np.dot(W[y[i]],X[i])

        f = f_left_term + f_right_term

        prob_c_given_XWT = np.zeros([k, n])
        for c in range(k):
            for i in range(n):
                prob_c_given_XWT[c, i] = exp_XWT[i, c] / sum_exp_XWT[i]

        G = np.zeros((k,d))
        for c in range(k):
            for j in range(d):
                g_left_factor = X[:, j]
                g_right_factor = prob_c_given_XWT[c, :] - (y == c)
                G[c, j] = np.sum(g_left_factor * g_right_factor)
        g = G.reshape(-1)
        return f, g
```

```python
# linear_models.py
class MulticlassLinearClassifier(LinearClassifier):
    """
    LinearClassifier's extention for multiclass classification.
    The constructor method and optimize() are inherited, so
    all you need to implement are fit() and predict() methods.
    """

    def fit(self, X, y):
        """YOUR CODE HERE FOR Q3.4"""
        n, d = X.shape
        k = len(np.unique(y))
        w_inital = np.zeros(k*d)

        w, fs, gs, ws = self.optimize(w_inital, X, y)
        W = w.reshape([k, d])
        self.W = W

    def predict(self, X_hat):
        """YOUR CODE HERE FOR Q3.4"""
        return np.argmax(X_hat @ self.W.T, axis=1)
```

3.5 Comparison with scikit-learn [2 points]

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. For one-vs-all, set `multi_class='ovr'`; for softmax, set `multi_class='multinomial'`. Since your comparison code above isn't using regularization, set `penalty='none'`. 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:
Our implementation

LinearClassifierOneVsAll training 0-1 error: 0.084
LinearClassifierOneVsAll validation 0-1 error: 0.070

SoftmaxLoss training 0-1 error: 0.000
SoftmaxLoss validation 0-1 error: 0.008

SKL implementation

SKL One-vs-All training 0-1 error: 0.086
SKL One-vs-All validation 0-1 error: 0.070

SKL SoftmaxLoss training 0-1 error: 0.000
SKL SoftmaxLoss validation 0-1 error: 0.016

The training and validation error for both one-vs-all model and SoftmaxLoss model is similar between our implementation and SKL's implementation, this shows our results are reliable, and that SKL could be implemented using similar algorithms.

main.py

```python
@handle("3.5")
def q3_5():
    from sklearn.linear_model import LogisticRegression

    data = load_dataset("multiData")
    X, y = data["X"], data["y"]
    X_valid, y_valid = data["Xvalid"], data["yvalid"]

    """YOUR CODE HERE FOR Q3.5"""
    skl_one_vs_all = LogisticRegression(fit_intercept=False, penalty=None,
     ↪  multi_class='ovr', max_iter=500)
    skl_softmax = LogisticRegression(fit_intercept=False, penalty=None,
     ↪  multi_class='multinomial', max_iter=500)
    skl_one_vs_all.fit(X,y)
    skl_softmax.fit(X,y)

    one_vs_all_train_err = classification_error(skl_one_vs_all.predict(X), y)
    one_vs_all_val_err = classification_error(skl_one_vs_all.predict(X_valid), y_valid)
    softmax_train_err = classification_error(skl_softmax.predict(X), y)
    softmax_val_err = classification_error(skl_softmax.predict(X_valid), y_valid)

    print(f"SKL One-vs-All training 0-1 error: {one_vs_all_train_err:.3f}")
    print(f"SKL One-vs-All validation 0-1 error: {one_vs_all_val_err:.3f}")
    print(f"model predicted classes: {np.unique(skl_one_vs_all.predict(X))}")
    print(f"SKL SoftmaxLoss training 0-1 error: {softmax_train_err:.3f}")
    print(f"SKL SoftmaxLoss validation 0-1 error: {softmax_val_err:.3f}")
    print(f"model predicted classes: {np.unique(skl_softmax.predict(X))}")
```

3.6 Cost of Multi-Class Logistic Regression [4 points]

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: Forming Z is $O(n^2d)$ and training on one iteration is $O(n^2k)$ so doing it for T iteration it is $O(n^2kT)$ therefore:

   $O(n^2d + n^2kT)$

2. What is the cost of classifying the t test examples?

   Answer: Computing $\tilde{Z}$ is $O(ndt)$ while computing $\tilde{Z}W$ is $O(tnk)$ therefore:

   $O(ndt + tnk)$

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 [18 points]

1. Suppose that a client wants you to identify the set of "relevant" factors that help prediction. Should you promise them that you can do this?

   Answer: No, feature selection algorithms and methods are not perfect and rely greatly on the data provided, samples, and input from experts on the subject matter. Those methods can help identify factors, but there is never a guarantee that only the relevant factors are identified.

2. What is a setting where you would use the L1-loss, and what is a setting where you would use L1-regularization?

   Answer: L1-loss is used when the model needs to be robust to outliers, whereas the L1-regularization is used when the model needs to be robust to irrelevant features (i.e. for feature selection as well.)

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

   Answer: L0: Non-Convex objectives, non-unique but sparse solutions.
   L1: Convex objectives, non-unique but sparse solutions.
   L2: Convex objectives, unique solutions but non-sparse.

4. 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: As λ increases, the sparsity level of the solution increases. As λ increases the training error also increases but the generalization gap decreases.

5. Suppose you have a feature selection method that tends not to 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: To reduce false negatives, the ensemble method is the bootstrapped samples method where firstly we generate bootstrap samples of the training data, run the feature selection method, then only take the features selected in all bootstrap samples

6. Suppose a binary classification dataset has 3 features. If this dataset is "linearly separable", what does this precisely mean in three-dimensional space?

   Answer: What this means is there that there exists a 2d plane that can completely separate all the examples such that all examples that are of "0" class are on one side of the plane and all the examples are of "1" are on the other side.

7. 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 number of classification errors is non-convex, thus does not converge to a minimum. Logistical loss is convex and allows convergence to a solution.

8. What is a disadvantage of using the perceptron algorithm to fit a linear classifier?

   Answer: The perceptron algorithm does not maximize the margin so it does not provide the optimal separation of samples, resulting in worse generalization for classifying unseen data.

9. How does the hyper-parameter σ affect the shape of the Gaussian RBFs bumps? How does it affect the fundamental tradeoff?

   Answer: Small σ results in a a skinny tall bump while large σ results in wider bumps. As σ decreases, the training error decreases but the model is more prone to overfitting so generalization gap increases.