CPSC 340 Assignment 4 (Due Friday 2022-03-11 at 11:59pm)

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 handwriting is *at all* difficult to read – at least these 5 points, more for egregious issues. Compared to assignment 1, your name and student number are no longer necessary (though it's not a bad idea to include them just in case, especially if you're doing the assignment with a partner).

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: It can be shown that the second derivative to this 1D equation is $f''(w) = 2\alpha$. Since $\alpha \geq 0$, this is always positive, and hence the original function is convex.

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

   Answer: It can be shown that the second derivative of this function is $f''(w) = \frac{1}{w^2}$. Since $w > 0$ is the domain of w, the second derivative is always positive, and hence $f(w)$ is convex.

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

   Answer: We learnt from class that all norms are convex. Thus, we know that $\|w\|_1$ is convex. Since $\lambda \geq 0$, we know that $\frac{\lambda}{2}\|w\|_1$ is convex too. The first term is a linear function $Xw - y$ inside a convex function, the L1-norm. By the composition rule, we know that $\|Xw - y\|_1$ is convex as well. Then, the sum of the two convex functions will also be 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: First, note that $-y_i w^T x_i$ is a constant in logistic regressions. Then, let $g(z) = \log(1+\exp(z))$, for a 1-D input z. It can be shown that the second derivative of $g(z)$ is such that $g''(z) = \frac{\exp(-z)}{(1+\exp(-z))^2}$. No matter the domain of z, the output of $\exp(-z)$ will always be positive, meaning that $g''(z) > 0$ for all z possible. Thus, it follows that $g(z)$ is a convex function. Since $-y_i w^T x_i$ is a constant, we know that $g(-y_i w^T x_i) = \log(1 + \exp(-y_i w^T x_i))$ is convex. Finally, this question has the sum from $i = 1$ to n of these terms, and the sum of convex functions is convex, so thus $f(w)$ is convex.

5. $f(w) = \sum_{i=1}^{n} [\max\{0, |w^T x_i - y_i|\} - \epsilon] + \frac{\lambda}{2}\|w\|_2^2$ with $w \in \mathbb{R}^d, \epsilon \geq 0, \lambda \geq 0$ (support vector regression).

   Answer: First, to examine the first term. It is the sum of $\max\{0, |w^T x_i - y_i|\} - \epsilon$ for all i from 1 to n. Each $|w^T x_i - y_i|$ is a linear equation that is within an absolute value function. Since the absolute value function is convex, their composition is convex. In class, we mentioned that the maximum of convex functions is convex, so since both 0 and $|w^T x_i - y_i|$ are convex, their max is also convex. By another

convex function rule, we then know that the sum of all these max terms from $i = 1$ to n will also be convex since each of them are convex. Also, note that subtracting by ϵ does not change the convexity.

Now, the second term. From the slides, we know that all squared norms are convex functions, so $\|w\|_2^2$ is convex. Since λ is a positive value, multiplying it keeps the term convex. Then, finally, the sum of these two terms (which are both individually convex) is also thus 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 -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 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 `LogRegClassifier` 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.

Note: as you may have noticed, `lambda` is a special keyword in Python, so we can't use it as a variable name. Some alternative options: `lammy`, `lamda`, `reg_wt`, λ if you feel like typing it, the sheep emoji ....

Answer:

The code is as below. This code pretty much directly calculates the loss function like the `LogisticRegressionLoss` class does, but adding on the norm squared term.

```
class LogisticRegressionLossL2(LogisticRegressionLoss):
    def __init__(self, lammy):
        super().__init__()
        self.lammy = lammy

```

```
    def evaluate(self, w, X, y):
        w = ensure_1d(w)
        y = ensure_1d(y)

        Xw = X @ w
        yXw = y * Xw

        # function value
        f = np.sum(np.log(1 + np.exp(-yXw)))
                    + self.lammy / 2 * np.square(np.linalg.norm(w))

        # gradient value
        loss = -y / (1+np.exp(yXw))
        grad_loss = X.T @ loss
        grad_reg = self.lammy * w
        g = grad_loss + grad_reg

        return f, g
```

The quantities reported with $\lambda = 1$ are as follows:

1. Training error: 0.002
2. Validation error: 0.074
3. # of features used: 101 (which is also the whole dataset)
4. # of function evaluations: 30

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 `LogRegClassifier` 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: My code here very much mirrors the code from the previous questions. The difference here is since the optimizer handles the regularization, I just instantiated the regular LogisticRegressionLoss() class and pass that in.

```
    # lammy = 0.01
    # lammy = 0.1
    # lammy = 1
    lammy = 10
```

```
    fun_obj = LogisticRegressionLoss()
    optimizer = GradientDescentLineSearchProxL1(lammy= lammy, max_evals=400,
    ↪                                                          verbose = False)
    model = linear_models.LogRegClassifier(fun_obj, optimizer)
    model.fit(X,y)
    print(f"With lambda = {lammy}:")
    train_error = utils.classification_error(model.predict(X), y)
    print(f"Log L1 Reg Training error: {train_error:.3f}")

    validation_error = utils.classification_error(model.predict(X_valid), y_valid)
    print(f"Log L1 Reg Validation error: {validation_error: .3f}")

    print(f"# nonZero features: {np.sum(model.w != 0)}")
    print(f"# iterations: {optimizer.num_evals}")
```

The errors/records I got were:

	$\lambda = 0.01$	$\lambda = 0.1$	$\lambda = 1$	$\lambda = 10$
Training Error	0	0	0	0.05
Validation Error	0.072	0.06	0.052	0.09
Num. Features Used	89	81	71	29
Num. GD Iterations	158	236	107	14

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 `LogRegClassifierForwardSel` 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: My code is as belows. The forward selection takes out the `X_subfeatures` with the `selected` features, then runs basic optimization on them, and updates the min/best_features. **The code that I wrote is line 24, and lines 25 - 31**.

```
    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=" ")

            best_feature = np.inf

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

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

                X_subfeatures = X[:,selected_with_j]
                X_sub_n, X_sub_d = X_subfeatures.shape

                w_init_guess = np.zeros(X_sub_d)
                w_opt, *_ = self.optimize(w_init_guess, X_subfeatures, y)

                local_loss, *_ = self.loss_fn.evaluate(w_opt, X_subfeatures, y)

                if local_loss < min_loss:
                    min_loss = local_loss
                    best_feature = j

                self.total_evals += self.optimizer.num_evals

            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
```

Using this code, I found the following:

1. Training error: 0
2. Validation error: 0.044
3. # Features selected: 27
4. # Total optimization steps: 209,459

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:

1. First, the effect of regularization on sparsity. We had learned about how L2 regularization doesn't really promote sparsity, L1 regularization promotes it, and L0 regularization **really** promotes sparsity. We can see that clearly here, where L2 reg. uses **all** 101 features, L1 reg. uses less features, and L0 reg. uses the least features (even with λ just being 1). This is a direct result of how the norms are calculated, and these were all mentioned in the lecture slides (about how the penalty for some $w_i \in w, w_i = 0.001$ would be $w_i^2 = 0.000001\lambda$, $|w_i| = 0.001\lambda$, and λ for L2, L1, and L0 regularization, respectively).
2. Validation error: Even though all three approaches were all relatively accurate, we can see that the L1 regularization produced better validation error than L2 reg., even though they are practically doing the same amount of calculations. This is probably because L1 regularization picks out the most important features more than L2-regularization. This is likely also the same reason why our L0-regularization has even better validation error.
3. Time/steps: Generally, it seemed like L1-regularization required more steps than L2-regularization. L0-regularization is not exactly modelled the same way, but we can imagine that it would require more computational effort.

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: The loss is $f(w) = \frac{1}{2}(w^2 - 4w + 5) + \lambda\sqrt{w}$.

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

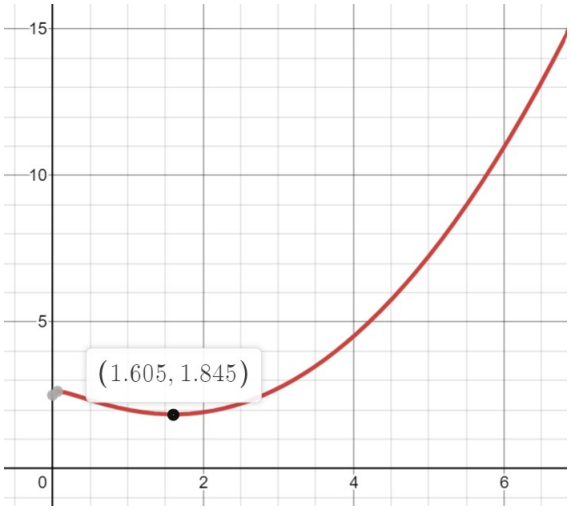
   Answer: If $\lambda = 0$, we can rearrange this into vertex form, finding that the $\arg\min_w f(w) = 2$.

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

   Answer: Since λ is directly part of the loss function, to minimize $f(w)$ would mean that $w = 0$.

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`.)

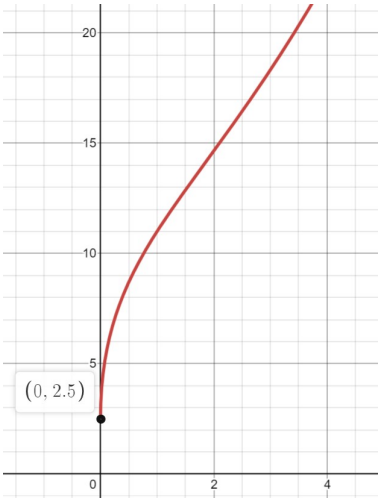
   Answer: Here is the plot:



   The $\arg\min$ is $w = 1.6$.

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:



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

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: $L\frac{1}{2}$ regularization behaves more like L1 regularization because it promotes sparsity much more

than L2, and in a way that is more similar to L1. This is because of the square-root penalty term. The potential penalty of $w_j = 0.01$ is actually $\sqrt{0.01} = 0.1\lambda$, so any values close to zero in w has a magnified penalty. The L2 regularization is the *opposite of this*, so L1 is relatively closer.

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

   Answer: No, it is not a convex optimization problem. In the simplest case as with this question, where $d = 1$, $n = 2$, we can take the second derivative of the loss function. $f''(w) = 1 + -\frac{1}{4}\lambda x^{-\frac{3}{2}}$. To be convex, this needs to be positive for its entire domain, but that is not the case here. Since I used desmos, I could actually visualize what the function was like, and for certain values of λ, it was definitely not convex.

3 Multi-Class Logistic Regression [32 points]

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 [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 $y_i \in \{1, 2, \ldots, k\}$. 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:

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

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

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

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

Answer:

We need to calculate the $w_c^T x_i$ for every class c. Here, we have three cases.

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

Thus, we can conclude that because $w_3^T \tilde{x}$ is the largest value, that we should predict class 3.

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 `LogRegClassifierOneVsAll` 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: My code for the `LogRegClassifierOneVsAll` class is as follows. The part that I wrote is between `lines 18-36`.

```
1     class LogRegClassifierOneVsAll(LogRegClassifier):
2
3         def fit(self, X, y):
4             n, d = X.shape
5             y_classes = np.unique(y)
6             k = len(y_classes)
7             assert set(y_classes) == set(range(k))  # check labels are {0, 1, ..., k-1}
8
9             # quick check that loss_fn is implemented correctly
10            self.loss_fn.check_correctness(np.zeros(d), X, (y == 1).astype(np.float32))
11
12            # Initial guesses for weights
13            W = np.zeros([k, d])
14
15            # NOTE: make sure that you use {-1, 1} labels y for logistic regression,
16            #       not {0, 1} or anything else.
17
18            y_each_class = np.zeros_like(y)
19            ## Train each class separately
20            for c in range(k):
21                ## Label all non-c class as -1, all c-class as 1
22                examples_of_class_c = y == c
23                y_each_class[examples_of_class_c] = 1
24                y_each_class[~examples_of_class_c] = -1
25
26                ## LogisticRegressionOptimize on it
27                w_c = np.zeros(d)
28                w_c_opt, *_ = self.optimize(w_c, X, y_each_class)
29
30                ## Store that w_c into the matrix at row c
31                W[c:c+1] = w_c_opt
32
33            self.W = W
34
35        def predict(self, X):
36            return np.argmax(X @ self.W.T, axis=1)
```

The validation error I received using this model was 0.07.

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: For rows w_j in W, there are two cases. Either you **are** the correct class ($y_i = j$) or you are not. Let's first examine the case that it w_j is the correct class. Also, since derivatives can be collected over a summation, I will look at one generic case in the summation, x_i.

Since w_j is the correct class, $w_{y_i} = w_j$.

$$\frac{\partial f}{\partial w_j} = -x_i + \frac{\exp(w_j^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)} * x_i$$

Once we can see that, it becomes easy to see that if w_j was NOT the correct class, we would get something very similar:

$$\frac{\partial f}{\partial w_j} = 0 + \frac{\exp(w_j^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)} * x_i$$

The only thing that changes is the first term, where it is either 0 or 1 multiplied by $-x_i$: 0 if it's **not** the correct class, and 1 if it is correct. Also, we can notice that the fraction term which arises from the chain rule, $\frac{\exp(w_j^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}$ is equal to $p(y_i = c \mid W, x_i)$ as defined in the question above.

Hence, by i) factoring out the x_i, ii) making it specific to the value in column j, iii) reordering some terms, iv) using the indicator function notation, and v) reintroducing the sum (not that the derivatives are not affected by it), it can be shown that:

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

3.4 Softmax Classifier Implementation [8 points]

Inside `linear_models.py`, you will find the class `MulticlassLogRegClassifier`, 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 the instructors are.

Answer: This answer came from some mathematical manipulations. I tried my best to not answer the question using the for-loops approach, hence I wanted to deduce the loss function and gradients in matrix form. Below is the derivation for it:

Based off the derivation in Q3.3, I knew that we probably needed to have every combination of $w_c^T x_i$ possible, as it would ended up being used in the probability statement. To do this, I first multiplied the X and W matricies. The notation I used was the following:

$$X = \begin{bmatrix} - - x_1^T - - \\ - - x_2^T - - \\ \vdots \\ - - x_n^T - - \end{bmatrix}, W = \begin{bmatrix} - - w_1^T - - \\ \vdots \\ - - w_c^T - - \end{bmatrix}$$

Now, we can multiply W by X^T, and we will get matrix M:

$$M = WX^T = \begin{bmatrix} - - w_1^T - - \\ \vdots \\ - - w_c^T - - \end{bmatrix} \times \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} w_1^T x_1 & \cdots & w_1^T x_n \\ \vdots & \ddots & \\ w_n^T x_1 & \cdots & w_n^T x_n \end{bmatrix}$$

Where $M[c, j] = w_c^T x_j$. Since this question will almost exclusively deal with $\exp(w_c^T x_j)$, I decided to raise the entire M matrix to the natural exponent, so our final M looks like:

$$M = \begin{bmatrix} \exp(w_1^T x_1) & \cdots & \exp(w_1^T x_n) \\ \vdots & \ddots & \\ \exp(w_n^T x_1) & \cdots & \exp(w_n^T x_n) \end{bmatrix}$$

Naturally, it follows that $p(y_i = c \mid W, x_i) = \dfrac{M[c, i]}{\sum_{j=1}^{n} M[c, j]}$. I wanted to make a matrix that pre-computed all of these probabilities so that we could just read them using indexes. This was called P, where P is a $k \times n$ matrix such that $P[c, j] = p(y_j = c \mid W, x_j)$. I constructed this with a for-loop in the code using the exact formula above.

Lastly, I also constructed another $k \times n$ matrix called Y where it replaces the indicator function. Specifically, I constructed Y such that $Y[c, j] = \{1 \text{ if } y_j == c, \text{ and } 0 \text{ otherwise}\}$. This was relatively simple and was done in the same for-loop to construct P.

Now, to examine the gradient matrix, W'. From previous questions, we know that:

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

By expanding the summation, we see that:

$$W'[c, j] = x_{1j}[p(y_1 = c \mid W, x_1) - \mathbb{1}(y_1 = c)] + \cdots + x_{nj}[p(y_n = c \mid W, x_n) - \mathbb{1}(y_n = c)]$$

In inner product form, we can rewrite this as:

$$\left\langle \begin{bmatrix} x_{1j} \\ \vdots \\ x_{nj} \end{bmatrix}, \begin{bmatrix} p(y_1 = c \mid W, x_1) - \mathbb{1}(y_1 = c) \\ \vdots \\ p(y_n = c \mid W, x_n) - \mathbb{1}(y_n = c) \end{bmatrix} \right\rangle = \left\langle \text{j-th column of } X, \begin{bmatrix} p(y_1 = c \mid W, x_1) \\ \vdots \\ p(y_n = c \mid W, x_n) \end{bmatrix} - \begin{bmatrix} \mathbb{1}(y_1 = c) \\ \vdots \\ \mathbb{1}(y_n = c) \end{bmatrix} \right\rangle$$

Finally, we reach the conclusion that:

$$W'[c, j] = \langle j\text{-th column of } X, c\text{-th row of } P - c\text{-th row of } Y \rangle$$

$$W' = (P - Y) \times X$$

There is less derivation necessary for calculating the loss function value, hence it will not be shown here. Since it uses a lot of the similar values, it will still use the matrix M for some (but not all) of its values. Here is my code for `class SoftmaxLoss` in `fun`$_o bj$.

```
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))

        ## f
        W = w.reshape((k,d))

        ## Calculating the Matrices M, P, Y

        ## Making M:
        ## M has properties such that M is k x d, and M[c,i] = exp(w_c.T @ x_i)
        M = np.exp(W @ X.T)

        ## Making P and Y (indicator)
        ## P has properties such that it is k x n, where:
        ## P[c,i] = P(y_i = c | x_i, W)

        ## Y has properties such that it is k x n, and:
        ## Y[c,i] = 1 if y_i == c, 0 if not.

        P = np.zeros([k,n])
```

```
        Y = np.zeros([k,n])

        for i in range(n):
            sum_of_all_probabilities_for_xi = np.sum(M[:, i])
            for c in range(k):
                prob_of_class_c = M[c,i]
                prob_final = prob_of_class_c/ sum_of_all_probabilities_for_xi
                P[c,i] = prob_final

                Y[c,i] = 1 if y[i] == c else 0

        ## Calculate f
        total_f = 0
        for i in range(n):
            y_i = y[i]
            w_yi_x_i = np.log(M[y_i, i])      ##because M is all exp() values
            inner_sum = np.sum(M[:, i])
            total_f += -1 * w_yi_x_i + np.log(inner_sum)

        ## Calculate g
        W_prime = (P - Y) @ X

        w_prime_flat = W_prime.flatten()

        return total_f, w_prime_flat
```

Then, here is my code for MulticlassLogRegClassifier in linear_models:

```
    class MulticlassLogRegClassifier(LogRegClassifier):
        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
            W_init = np.zeros([k, d])
            W_init = W_init.reshape(-1)
            self.loss_fn.check_correctness(W_init, X, y)

            W = np.zeros([k,d])
            W = W.reshape(-1)

            W_opt, *_ = self.optimize(W, X, y)
            W_opt = W_opt.reshape((k,d))

            self.W = W_opt

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

The validation error I received was 0.008.

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'`, or just set `C` very large to effectively disable regularization. (Remember that, basically, $C = 1/\lambda$.) 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:

For OneVsAll implementation, both training error and validation error were exactly the same. Training error was 0.084 and validation error was 0.07 for both models.

For Multi-Class Implementation, for both models, training error was 0. For validation error, SciKit-Learn's model was 0.016 and ours was 0.008.

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: For training the data:

   (a) Transform X into Z. To do this, we must compare all n^2 pairs of datapoints by applying the function $g(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|_2^2}{2\sigma})$. Each calculation will take $O(d)$ time, and there are n^2 of them, totalling $O(n^2 d)$ time.

   (b) Once the points are transformed into Z, we can now start doing the gradient descent. There are T total gradient descent steps, and each step requires the following work (in my implementation, at least):

      i. Calculate $M = W \times Z^T$. This is multiplying a $(k \times n)$ matrix by a $(n \times n)$ matrix, creating a matrix of size $k \times n$. Each entry in the matrix requires an inner product between two vectors of length n. Therefore, the total work here is $n \times (k \times n) = O(n^2 k)$.

      ii. Making P, Y matricies. These are constructed in one for loop (see Q3.4) that creates both $k \times n$ matricies based off the information in M. Specifically, it conducts $k \times n$ index reads and n calculations of the sum of one column of M. This adds up to a total of $O(n^2 + nk)$ time.

      iii. Calculating f. The loss function requires a sum of a certain calculation across all n data points. That calculation requires $O(n)$ time, totalling to $O(n^2)$ time.

      iv. Calculating g. The purpose of all the work for making matricies M, P, Y was so that this step was fast(er). Here, we are calculating $(P - Y)Z$, which creates a $k \times n$ matrix whose each entry requires n calculations, totalling $O(n^2 k)$ time.

Hence, we can see that each iteration of gradient descent takes $O(n^2k)$ time, which totals $O(n^2kT)$ time overall.

Therefore, in total, it takes $O(n^2(d + kT))$ time to train this model.

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

   Answer: To test the data:

   (a) First, transform t testing examples into $\tilde{Z}$ using training data points. To do this, we create a matrix of size $t \times n$ where each entry requires d time to calculate. This adds up to $O(ndt)$.

   (b) Then, to predict. To predict, we need to multiply $\tilde{Z} \times W^T$ to get our result matrix of size $t \times k$. Each entry in this matrix requires n calculations. Thus, this totals $O(nkt)$ time. Then, to classify, we need to apply the arg max function across each row of the matrix. This requires k time for each row, and there are t total rows, totalling $O(kt)$. The total time for this step is still $O(nkt)$.

   Thus, the total running time of classifying t test examples of $O(nt(d + k))$.

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]

Answer each of the following questions in a sentence or two.

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: In general, **no I wouldn't promise it**. We have seen in this class that there exist many different approaches to appropriate feature selection, but all of them have their downfalls (and perhaps why so many exist to begin with). I think I would tell the client that we can probably achieve a set of good features to use, but I think "relevance" is hard to quantify when it comes to modeling.

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 robust to outliers, so when you're not sure where/how to find the outliers in a dataset but still want to apply regression; L1-regularization is robust to irrelevant features, allowing you to develop relatively sparse weight vectors and thus indirectly selecting the important features.

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

   Answer:

   (a) Convex: Both L1-regularization and L2-regularization are convex objectives. L0 is not.

   (b) Uniquity: Only L2-regularization yields a unique solution

   (c) Sparse solutions: L1-regularization and L0-regularization produce sparse solutions. L2-regularization does not.

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: A higher λ means a higher penalty on the size of the weight vector. Hence, larger λ leads to more sparse solutions. Thus, we can think of λ as inversely proportional to the model's complexity. In that way, lower λ values would decrease E_{train} but increase E_{approx} (as it is less generalizable) and higher λ values would increase E_{train} but decrease E_{approx} as it is theoretically becoming more generalizable.

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: I would first run a linear classifier with L1-regularization on the data first, because L1-regularization tends to capture the relevant features but keeps many false positives. Then, I would then pass the features selected by the L1-regularized weight vector through this feature selection method, ideally creating a good group of selected features at the end.

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: It means that in the 3-dimensional data-space, there exists a hyperplane that can perfectly separate the datapoints of the two classes.

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: This is because the logistic loss is convex and differentiable, making it a lot easier to optimize. Also, we talked about in class how logistic loss has a direct probabilistic interpretation that allows it to make sense to us when creating linear classifiers.

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

   Answer: It only guarantees a perfect classifier if the data is linearly separable, and it seems to take a long time to train for what could be done with more effective methods.

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

   Answer: The σ is the width of the "bumps". By using smaller σ, we will be using "thinner" Gaussian RBF bumps, which will make the regression line much less smooth and have more turning points (since the transition between bumps may not be as smooth). This will make the model more fine-tuned to the training data provided (therefore decreasing E_{train}), but make it not as generalizable, especially at points which are **between** the bumps, thus increasing E_{approx}. The opposite is true for a larger σ.