CPSC 340 Assignment 4 (Due 2023-11-10 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 submission is *at all* difficult to read – at least these 5 points, more for egregious issues.

Credits: Used chatGPT and lecture notes for reference across the entire assignment.

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:

   $$f'(w) = 2\alpha w - \beta$$
   $$f''(w) = 2\alpha \quad (\text{since } \alpha > 0, f''(w) >= 0)$$

   Hence, this is a convex function .

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

   Answer:

   $$f'(w) = -\frac{1}{w}$$
   $$f''(w) = \frac{1}{w^2} \quad (\text{since } \alpha > 0 \text{ and } w^2 \text{ is non-negative } f''(w) >= 0)$$

   Hence, this 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).

   Answer:

Given:	The norm function is convex.
First part:	$\|Xw - y\|_1$ is convex as it is a norm function.
Second part:	$\|w\|_1$ is convex, and multiplying a convex function by a constant preserves its convexity.
Conclusion:	The sum of convex functions is convex. Therefore, $\|Xw - y\|_1 + \|w\|_1$ is a convex function.

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:

   To demonstrate the convexity of one term of $\log(1 + \exp(...))$,
   we treat $\exp(-y_i w^T x_i)$ as z. Then:

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

   $$f''(w) = \frac{\exp(z)}{(1 + \exp(z))^2}.$$

   Since the numerator is always positive and the square of the
   denominator is non-negative, 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).

   Answer:

Given:	The norm function is convex, and linear function is convex
First part:	$\|w^T x_i - y_i\|$ is a linear function. 0 is a linear function and ϵ is a constant, so the sum of the first part is a convex function
Second part:	The norm is convex, and the composition of a constant and a convex is convex
Conclusion:	f is convex

General hint: for the first two you can check that the second derivative is non-negative since they are one-dimensional. For the last 3, 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.

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` (what Mike's niece calls her toy stuffed lamb), `lamda`, `reg_wt`, λ if you feel like typing it, the sheep emoji[1], ...

Answer:
The training error goes from 0.000 to 0.002
The validation error goes from 0.082 to 0.074
The non zeros is the same – 101

[1] Harder to insert in LaTeX than you'd like; turns out there are some drawbacks to using software written in 1978.

The function evals goes from 89 to 30

```python
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)

        """YOUR CODE HERE FOR Q2.1"""
        Xw = X @ w
        yXw = y * Xw  # element-wise multiply; the y_i are in {-1, 1}

        #--------
        logistic_loss = np.sum(np.log(1 + np.exp(-yXw)))

        # Add regularization term to the loss
        regularization_loss = 0.5 * self.lammy * np.sum(w ** 2)
        total_loss = logistic_loss + regularization_loss

        # Compute the gradient
        # Calculate the term for the gradient of logistic loss
        logistic_grad = -y / (1 + np.exp(yXw))

        # Calculate the total gradient including the regularization term
        gradient = X.T @ logistic_grad + self.lammy * w
        return total_loss, gradient
```

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.[2]

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.

[2]Here's an explanation, as **bonus content** you don't need to understand.
(Feel free to delete this overly long footnote from your answers file, if you want.)
For the explanation to make sense, it'll help to first re-frame gradient descent in the following way: to take a step from w^t while trying to minimize an objective function f, we first make a *quadratic approximation* to f around the point w^t of the form

$$\tilde{f}^t(w) = f(w^t) + [\nabla f(w^t)]^T(w - w^t) + \frac{1}{2\alpha^t}\|w - w^t\|^2.$$

This is like taking a Taylor expansion of f, but instead of using the expensive-to-compute Hessian $\nabla^2 f$, we just use $\frac{1}{\alpha^t} I$. Then we minimize that approximation to find our next step: $w^{t+1} = \arg\min_w \tilde{f}^t(w)$, which if you do out the math ends up being exactly our old friend $w^{t+1} = w - \alpha^t \nabla f(w^t)$.[3]
In proximal gradient descent, our objective $f(w)$ is of the form $g(w) + h(w)$, where g is a smooth function (e.g. the logistic regression loss) but h might not be differentiable. Then the idea of proximal gradient descent is that we do the quadratic approximation for g but just leave h alone:

$$\begin{aligned} w^{t+1} &= \arg\min_w g(w^t) + [\nabla g(w^t)]^T(w - w^t) + \frac{1}{2\alpha^t}\|w - w^t\|^2 + h(w) \\ &= \arg\min_w \frac{1}{2\alpha^t}\|w - (w^t - \alpha^t \nabla g(w))\|^2 + h(w), \end{aligned} \tag{prox}$$

an optimization problem trying to trade off being close to the gradient descent update (first term) with keeping h small (second).
As long as you can compute $\nabla g(w)$, this problem otherwise *doesn't depend on g at all*: you can just run the gradient descent update based on g then plug that into the "prox update" (prox). For many important functions h, this is available in closed form. For L1 regularization we have $h(w) = \lambda\|w\|_1$, and it turns out that the solution is the "soft-thresholding" function, given elementwise by

$$\left[\arg\min_w \frac{1}{2\alpha}\|w - z\|^2 + \lambda\|w\|_1\right]_i = \begin{cases} z_i - \alpha\lambda & \text{if } z_i > \alpha\lambda \\ 0 & \text{if } |z_i| \leq \alpha\lambda \\ z_i + \alpha\lambda & \text{if } z_i < -\alpha\lambda \end{cases}.$$

[3]Incidentally, using the real Hessian here is called Newton's method. This is a much better approximation to f, and so the update steps it takes can be much better than gradient descent, causing it to converge in many fewer iterations. But each of these iterations is much more computationally expensive, since we need to compute and solve a linear system with the $d \times d$ Hessian. In ML settings it's often too computationally expensive to run.[4]

```
dhcp-206-87-220-29:code simonxia$ python3 main.py 2.2
lambda = 0.01
Training error: 0.000
Validation error: 0.072
# feature used: 89
# gradient descent iteration: 158


lambda = 0.1
Training error: 0.000
Validation error: 0.060
# feature used: 81
# gradient descent iteration: 236


lambda = 1.0
Training error: 0.000
Validation error: 0.052
# feature used: 71
# gradient descent iteration: 107


lambda = 10.0
Training error: 0.050
Validation error: 0.090
# feature used: 29
# gradient descent iteration: 14
```

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

    """YOUR CODE HERE FOR Q2.2"""
    for lambda_value in [0.01, 0.1, 1, 10]:
        print(f"lambda = {lambda_value}")
        optimizer = GradientDescentLineSearchProxL1(max_evals=400, verbose=False, lammy=lambda_value)
        model = linear_models.LinearClassifier(LogisticRegressionLoss(), optimizer)
        model.fit(X, y)
        train_error = classification_error(model.predict(X), y)
        print(f"Training error: {train_error:.3f}")
        val_error = classification_error(model.predict(X_valid), y_valid)
        print(f"Validation error: {val_error:.3f}")
        print(f"# feature used: {np.sum(model.w != 0)}")
        print(f"# gradient descent iteration: {optimizer.num_evals}")
        print("\n")
```

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.

```
Linear training 0-1 error: 0.000
Linear validation 0-1 error: 0.038
# nonZeros: 24
total function evaluations: 145,502
```

```
    w_ini = np.zeros(selected_with_j.sum())
    w, _, _, _= self.optimize(w_ini, X[:, selected_with_j], y)
    self.total_evals += self.optimizer.num_evals
    loss, _ = self.global_loss_fn.evaluate(w, X[:, selected_with_j], y)
    if loss < min_loss:
        min_loss = loss
        best_feature = j
    #pass
```

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: L0 has the most expensive run time and L2 has a better validation erro than L1's, which is better than L0's. As L2 is the least sparse.

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

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

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

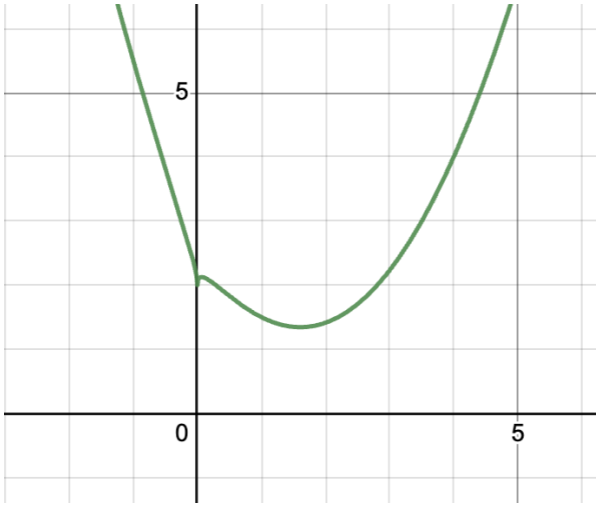
2. If $\lambda = 0$, what is the solution, i.e. $\arg\min_w f(w)$?

   Answer: w = 2

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

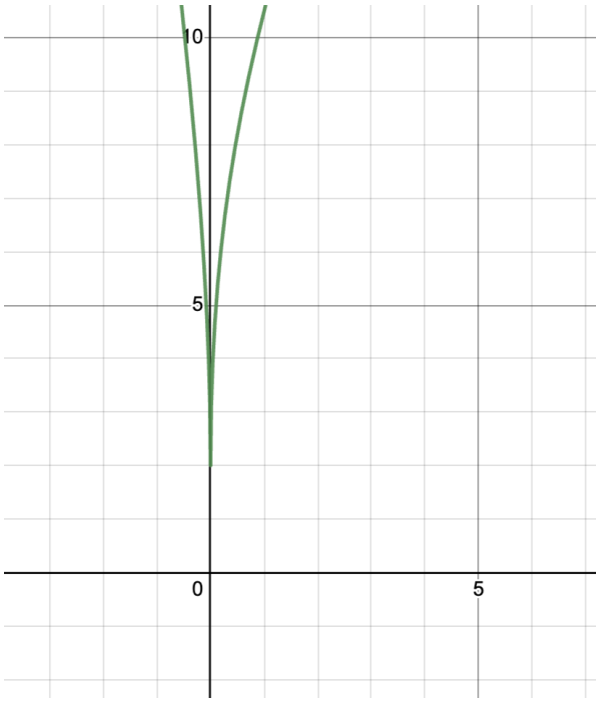
   Answer: w = 0

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: x =1.5

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: $x = 0$

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: L1 as some w are be zero

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

   Answer: No, not differentiable in points so not convex

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 $y_i \in \{1, 2, \ldots, k\}$. 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: Class 1: 2*1 + (-1)*1 = 1
Class 2: 2*1 + (-2)*1 = 0
Class 3: 3*1 + (-1)*1 = 2
$2 > 1 > 0$
Hence, we select 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 `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.

```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 i in range(k):
            # Convert labels for logistic regression: 1 for class i, -1 for others
            ytmp = y.copy().astype(float)
            ytmp[y == i] = 1
            ytmp[y != i] = -1
            # Initial guess
            w_init = np.zeros(d)
            w, _, _, _ = self.optimize(w_init, X, ytmp)
            W[i] = w
        self.W = W

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

Answer: The validation error is 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:
$f'(W) = \sum_{i=1}^{n}[\, -I(y_i = c) * x_{ij} + \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)} x_{ij}]$
For the second term, given $p(y_i = c \mid W, x_i) = \frac{\exp(w_c^T x_i)}{\sum_{c'=1}^{k} \exp(w_{c'}^T x_i)}$
$f'(W) = \sum_{i=1}^{n}[-I(y_i = c) * x_{ij} + p(y_i = c \mid W, x_i) * x_{ij}]$
Hence, $f'(W) = \sum_{c'=1}^{k} x_{ij} * [-I(y_i = c) + p(y_i = c \mid W, x_i)$

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.[5]

```python
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))
        w = w.reshape(k, d)

        # Compute the class scores for all data points
        scores = np.dot(X, w.T)  # Shape: [n, k]

        # Prevent numerical overflow by subtracting max for each row
        scores -= np.max(scores, axis=1, keepdims=True)

        # Compute the softmax probabilities
        exp_scores = np.exp(scores)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)  # Shape: [n, k]

        # Compute the loss
        correct_log_probs = -np.log(probs[np.arange(n), y])
        f = np.sum(correct_log_probs)

        # Compute the gradient
        dscores = probs
        dscores[np.arange(n), y] -= 1
        g = np.dot(dscores.T, X).reshape(-1)

        return f, g
```

Answer: The validation error obtained is 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'`. 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: Scikit-learn for One-vs-All has Training Error: 0.084, Validation Error: 0.07.
Scikit-learn for Softmax has Training Error: 0.0, Validation Error: 0.016

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

Assume that we have

- n training examples.
- d features.

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

- 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 new features of Z costs $d * n^2$,
   and gradient descent may cost ndk,
   and there could be T iteration. Hence,
   the sum is O($n^2 * d + ndkT$)

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

   Answer:

   The cost of evaluating $\hat{Z}$ involves computing the distance function t times.
   Each computation of the distance function costs nd.
   Therefore, the total computational cost is $O(ndt)$.

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: No, because feature selection is a messy topic and I can only try my best, and also "relevance" is not specific.

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

   Answer: If I want to select some features while being robust to outliers, I would use L1-Regularization. For L-1 Loss, maintaining robustness to outliers, it is used when the goal is to minimize the absolute differences between the predicted and actual values, and robustness against outliers is desired.

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

   Answer:
   For convexity: L1 and L2.
   For uniqueness: L2.
   For sparsity: L0 and L1.

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 goes up, the sparsity increases, and the training error goes up too. 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: Bootstrap sample of a list and include the features in a broader inclusion criterion as long as it appears once

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: Classes can be "separated" by a hyper-plane, so that one side of a plane is in a class and the other side is from the rest of the class

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: The non-convex characteristics couldn't use gradient descent to minimize w to find the best value

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

   Answer: The most significant limitation of the perceptron algorithm is that it can only converge to a solution if the data is linearly separable. If the training data cannot be separated by a linear boundary, the perceptron algorithm will fail to converge to a solution, leading to an endless cycle of adjustments.

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

   Answer: Affect the width of the bumps. Large σ yields high training error and smaller generalization gap.