CPSC 340 Assignment 3 (due Friday February 17 at 11:55pm)

Important: Please make sure to follow the submission instructions posted on Piazza.

Name(s) and Student ID(s):

1 More Unsupervised Learning

1.1 Vector Quantization

Discovering object groups is one motivation for clustering. Another motivation is *vector quantization*, where we find a prototype point for each cluster and replace points in the cluster by their prototype. If our inputs are images, we could use vector quantization on the set of RGB pixel values as a simple image compression algorithm.

Your task is to implement this simple image compression algorithm by writing a `quantizeImage` and a `deQuantizeImage` function. The `quantizeImage` function should take the name of an image file (like "dog.png" for the provided image) and a number b as input. It should use the pixels in the image as examples and the 3 colour channels as features, and run k-means clustering on this data with 2^b clusters. The code should store the cluster means and return four arguments: the cluster assignments y, the means W, the number of rows in the image $nRows$, and the number of columns $nCols$. The `deQuantizeImage` function should take these four arguments and return a version of the image (the same size as the original) where each pixel's original colour is replaced with the nearest prototype colour.

To understand why this is compression, consider the original image space. Say the image can take on the values $0, 1, \ldots, 254, 255$ in each colour channel. Since $2^8 = 256$ this means we need 8 bits to represent each colour channel, for a total of 24 bits per pixel. Using our method, we are restricting each pixel to only take on one of 2^b colour values. In other words, we are compressing each pixel from a 24-bit colour representation to a b-bit colour representation by picking the 2^b prototype colours that are "most representative" given the content of the image. So, for example, if $b = 6$ then we have 4x compression.

1. Complete the functions `quantizeImage` and `deQuantizeImage` in `vector_quantization.py` and hand in your code.
2. If you run `python main.py 1` it will run your compression algorithm on the included image 'dog.png' using a 1, 2, 4, and 6 bit quantization per pixel (instead of the original 24 bits). Hand in the resulting images obtained with this encoding.

2 Matrix Notation and Minimizing Quadratics

2.1 Converting to Matrix/Vector/Norm Notation

Using our standard supervised learning notation (X, y, w) express the following functions in terms of vectors, matrices, and norms (there should be no summations or maximums).

1. $\sum_{i=1}^{n} |w^T x_i - y_i| + \lambda \sum_{j=1}^{d} |w_j|$.
2. $\sum_{i=1}^{n} v_i (w^T x_i - y_i)^2 + \sum_{j=1}^{d} \lambda_j w_j^2$.
3. $\left(\max_{i \in \{1,2,\ldots,n\}} |w^T x_i - y_i|\right)^2 + \frac{1}{2} \sum_{j=1}^{d} \lambda_j |w_j|$.

You can use V to denote a diagonal matrix that has the (non-negative) "weights" v_i along the diagonal. The value λ (the "regularization parameter") is a non-negative scalar. You can use Λ as a diagonal matrix that has the (non-negative) λ_j values along the diagonal.

2.2 Minimizing Quadratic Functions as Linear Systems

Write finding a minimizer w of the functions below as a system of linear equations (using vector/matrix notation and simplifying as much as possible). Note that all the functions below are convex so finding a w with $\nabla f(w) = 0$ is sufficient to minimize the functions (but show your work in getting to this point).

1. $f(w) = \frac{1}{2}\|w - u\|^2$ (projection of u onto real space).
2. $f(w) = \frac{1}{2}\sum_{i=1}^{n} v_i(w^T x_i - y_i)^2 + \lambda w^T u$ (weighted and tilted least squares).
3. $f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w - w^0\|^2$ (least squares shrunk towards non-zero w^0).

Above we assume that u and w^0 are d-by-1 vectors and that v is a n-by-1 vector. You can use V as a diagonal matrix containing the v_i values along the diagonal.

Hint: Once you convert to vector/matrix notation, you can use the results from class to quickly compute these quantities term-wise. As a sanity check for your derivation, make sure that your results have the right dimensions. In order to make the dimensions match you may need to introduce an identity matrix. For example, $X^T Xw + \lambda w$ can be re-written as $(X^T X + \lambda I)w$.

2.3 Convex Functions

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.

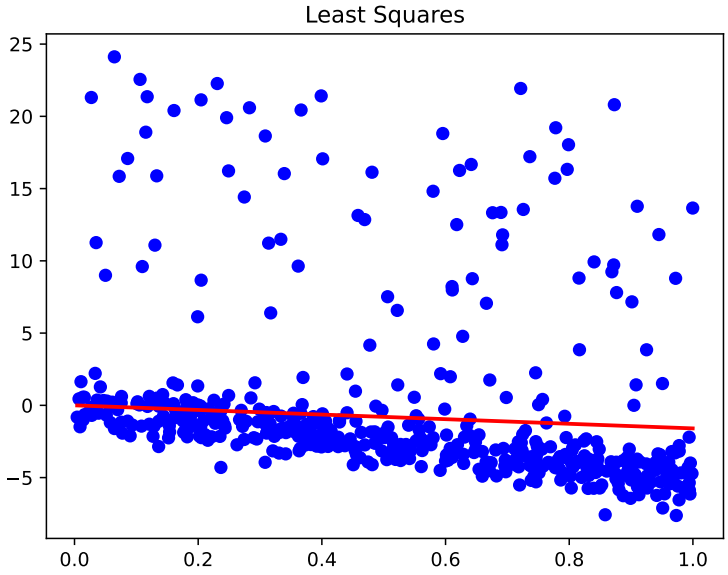
Show that the following functions are convex:

1. $f(w) = \frac{1}{2}w^2 + w^{-1}$ with $w > 0$.
2. $f(w) = \max_i w_i$ with $w \in \mathbb{R}^n$ (maximum).
3. $f(y) = \max(0, 1 - t \cdot y)$ with $y \in \mathbb{R}$ and $t \in \{-1, +1\}$ (hinge loss).
4. $f(w) = \|Xw - y\|^2 + \lambda\|w\|_1$ with $w \in \mathbb{R}^d, \lambda \geq 0$ (L1-regularized least squares).
5. $f(w) = \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i))$ with $w \in \mathbb{R}^d$ (logistic regression).

Hint for 2.3.5: this function may seem non-convex since it contains $\log(z)$ and log is concave, but there is a flaw in that reasoning: for example $\log(\exp(z)) = z$ is convex despite containing a log. To show convexity, it may be helpful to show that $\log(1+\exp(z))$ is convex, which can be done by computing the second derivative. It may simplify matters to note that $\frac{\exp(z)}{1+\exp(z)} = \frac{1}{1+\exp(-z)}$.

3 Robust Regression and Gradient Descent

If you run `python main.py 3`, it will load a one-dimensional regression dataset that has a non-trivial number of 'outlier' data points. These points do not fit the general trend of the rest of the data, and pull the least squares model away from the main downward trend that most data points exhibit:



Note: we are fitting the regression without an intercept here, just for simplicity of the homework question. In reality one would rarely do this. But here it's OK because the "true" line passes through the origin (by design). In Q4.1 we'll address this explicitly.

A coding note: when we're doing math, we always treat y and w as column vectors, i.e. if we're thinking of them as matrices, then shape $n \times 1$ or $d \times 1$, respectively. This is also what you'd usually do when coding things in, say, Matlab. It is *not* what's usually done in Python machine learning code, though: we usually have `y.shape == (n,)`, i.e. a one-dimensional array. Mathematically, these are the same thing, but if you mix between the two, you can really easily get confusing answers: if you add something of shape `(n, 1)` to something of shape `(n,)`, then the NumPy broadcasting rules give you something of shape `(n, n)`. This is a very unfortunate consequence of the way the broadcasting rules work. If you stick to either one, you generally don't have to worry about it; **we're assuming shape `(n,)` here**. Note that you can ensure you have something of shape `(n,)` with the `utils.ensure_1d` helper, which basically just uses `two_d_array.squeeze(1)` (which checks that the axis at index 1, the second one, is length 1 and then removes it). You can go from `(n,)` to `(n, 1)` with, for instance, `one_d_array[:, np.newaxis]` (which says "give me the whole first axis, then add another axis of length 1 in the second position").

3.1 Weighted Least Squares in One Dimension

One of the most common variations on least squares is *weighted* least squares. In this formulation, we have a weight v_i for every training example. To fit the model, we minimize the weighted squared error,

$$f(w) = \frac{1}{2}\sum_{i=1}^{n} v_i(w^T x_i - y_i)^2.$$

In this formulation, the model focuses on making the error small for examples i where v_i is high. Similarly, if v_i is low then the model allows a larger error. Note: these weights v_i (one per training example) are completely different from the model parameters w_j (one per feature), which, confusingly, we sometimes also call "weights." The v_i are sometimes called *sample weights* or *instance weights* to help distinguish them.

Complete the model class, `WeightedLeastSquares` (inside `linear_models.py`), to implement this model. (Note that Q2.2. asks you to show how a similar formulation can be solved as a linear system.) Apply this model to the data containing outliers, setting $v = 1$ for the first 400 data points and $v = 0.1$ for the last 100 data points (which are the outliers). Hand in your code and the updated plot.

3.2 Smooth Approximation to the L1-Norm

Unfortunately, we typically do not know the identities of the outliers. In situations where we suspect that there are outliers, but we do not know which examples are outliers, it makes sense to use a loss function that is more robust to outliers. In class, we discussed using the Huber loss,

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

where

$$h(r_i) = \begin{cases} \frac{1}{2} r_i^2 & \text{for } |r_i| \leq \epsilon \\ \epsilon(|r_i| - \frac{1}{2}\epsilon) & \text{otherwise} \end{cases}.$$

This is less sensitive to outliers than least squares, although it can no longer be minimized by solving a linear system. Derive the gradient ∇f of this function with respect to w. You should show your work but you do not have to express the final result in matrix notation. Hint: you can start by computing the derivative of h with respect to r_i and then get the gradient using the chain rule. You can use $\text{sgn}(r_i)$ as a function that returns 1 if r_i is positive and -1 if it is negative.

3.3 Gradient Descent: Understanding the Code

Recall gradient descent, a derivative-based optimization algorithm that uses gradients to navigate the parameter space until a locally optimal parameter is found. In `optimizers.py`, you will see our implementation of gradient descent, taking the form of a class named `GradientDescent`. This class has a similar design pattern as PyTorch, a popular differentiable programming and optimization library. One step of gradient descent is defined as

$$w^{t+1} = w^t - \alpha^t \nabla_w f(w^t).$$

Look at the methods named `get_learning_rate_and_step()` and `break_yes()`, and answer each of these questions, one sentence per answer:

1. Which variable is equivalent to α^t, the step size at iteration t?
2. Which variable is equivalent to $\nabla_w f(w^t)$ the current value of the gradient vector?
3. Which variable is equivalent to w^t, the current value of the parameters?
4. What is the method `break_yes()` doing?

3.4 Robust Regression

The class `LinearModel` is like `LeastSquares`, except that it fits the least squares model using a gradient descent method. If you run `python main.py 3.4` you'll see it produces the same fit as we obtained using the normal equations.

The typical input to a gradient method is a function that, given w, returns $f(w)$ and $\nabla f(w)$. See `fun_obj.py` for some examples. Note that the `fit` function of `LinearModel` also has a numerical check that the gradient code is approximately correct, since implementing gradients is often error-prone.[1]

3.4.1 Implementing the Huber Loss

An advantage of gradient-based strategies is that they are able to solve problems that do not have closed-form solutions, such as the formulation from section 3.2. The class `LinearModel` has most of the implementation of a gradient-based strategy for fitting the robust regression model under the Huber loss.

Optimizing robust regression parameters is the matter of implementing a function object and using an optimizer to minimize the function object. The only part missing is the function and gradient calculation inside `fun_obj.py`. Inside `fun_obj.py`, complete `RobustRegressionLoss` to implement the objective function and gradient function based on the Huber loss from section 3.2. Hand in your code and a regression plot using this robust regression approach with $\epsilon = 1$.

[1] Sometimes the numerical gradient checker itself can be wrong. See CPSC 303 for a lot more on numerical differentiation.

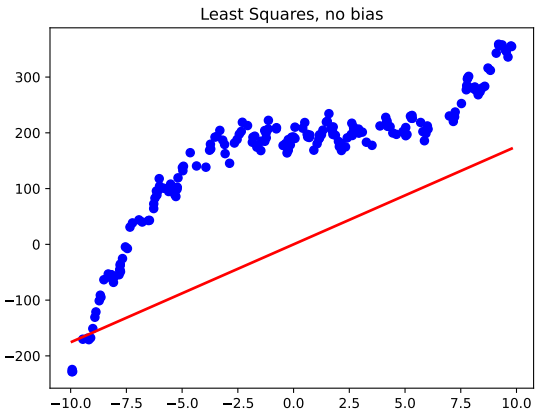
4 Linear and Nonlinear Regression

In class we discussed fitting a linear regression model by minimizing the squared error. In this question, you will start with a data set where least squares performs poorly. You will then explore how adding a bias variable and using nonlinear (polynomial) bases can drastically improve the performance. You will also explore how the complexity of a basis affects both the training error and the validation error.

If you run `python main.py 4`, it will:

1. Load a one-dimensional regression dataset.
2. Fit a least-squares linear regression model.
3. Report the training error.
4. Report the validation error.
5. Draw a figure showing the training data and what the linear model looks like.

Unfortunately, this is an awful model of the data. The average squared training error on the data set is over 28000 (as is the validation error), and the figure produced by the demo confirms that the predictions are usually nowhere near the training data:



4.1 Linear Regression with Bias Variable

The y-intercept of this data is clearly not zero (it looks like it's closer to 200), so we should expect to improve performance by adding a *bias* variable, so that our model is

$$y_i = w^T x_i + w_0$$

instead of

$$y_i = w^T x_i.$$

In file `linear_models.py`, complete the class `LeastSquaresBias`, that has the same input/model/predict format as the `LeastSquares` class, but that adds a *bias* variable w_0. Hand in your new class, the updated plot, and the updated training/validation error.

Hint: recall that adding a bias w_0 is equivalent to adding a column of ones to the matrix X. Don't forget that you need to do the same transformation in the `predict` function.

4.2 Linear Regression with Polynomial Basis

Adding a bias variable improves the prediction substantially, but the model is still problematic because the target seems to be a *non-linear* function of the input. Complete `LeastSquaresPoly` class, that takes a data vector x (i.e., assuming we only have one feature) and the polynomial order p. The function should perform a least squares fit based on a matrix Z where each of its rows contains the values $(x_i)^j$ for $j = 0$ up to p. E.g., `LeastSquaresPoly.fit(x,y)` with $p = 3$ should form the matrix

$$Z = \begin{bmatrix} 1 & x_1 & (x_1)^2 & (x_1)^3 \\ 1 & x_2 & (x_2)^2 & (x_2)^3 \\ \vdots & & & \\ 1 & x_n & (x_n)^2 & (x_n)^3 \end{bmatrix},$$

and fit a least squares model based on it. Submit your code, and a plot showing training and validation error curves for the following values of p: $0, 1, 2, 3, 4, 5, 10, 20, 30, 50, 75, 100$. Clearly label your axes, and use a logarithmic scale for y by `plt.yscale("log")` or similar, so that we can still see what's going on if there are a few extremely large errors. Explain the effect of p on the training error and on the validation error.

Note: large values of p may cause numerical instability. Your solution may look different from others' even with the same code depending on the OS and other factors. As long as your training and validation error curves behave as expected, you will not be penalized.

Note: you should write the code yourself; don't use a library like sklearn's `PolynomialFeatures`.

Note: in addition to the error curves, the code also produces a plot of the fits themselves. This is for your information; you don't have to submit it.

5 Very-Short Answer Questions

1. Describe a dataset with k clusters where k-means cannot find the true clusters.
2. Why do we need random restarts for k-means but not for density-based clustering?
3. Why is it not a good idea to create an ensemble out of multiple k-means runs with random restarts and, for each example, output the mode of the label assignments (voting)?
4. For each outlier detection method below, list an example method and a problem with identifying outliers using this method:
   - Model-based outlier detection.
   - Graphical-based outlier detection.
   - Supervised outlier detection.
5. Why do we minimize $\frac{1}{2}\sum_{i=1}^{n}(wx_i - y_i)^2$ instead of the actual mean squared error $\frac{1}{n}\sum_{i=1}^{n}(wx_i - y_i)^2$ in (1D) least squares?
6. Give an example of a feature matrix X for which the least squares problem *cannot* be solved as $w = (X^\top X)^{-1}(X^\top y)$.
7. Why do we typically add a column of 1 values to X when we do linear regression? Should we do this if we're using decision trees?
8. When should we consider using gradient descent to approximate the solution to the least squares problem instead of exactly solving it with the closed form solution?
9. If a function is convex, what does that say about stationary points of the function? Does convexity imply that a stationary points exists?
10. For robust regression based on the L1-norm error, why can't we just set the gradient to 0 and solve a linear system? In this setting, why we would want to use a smooth approximation to the absolute value?
11. What is the problem with having too small of a learning rate in gradient descent?
12. What is the problem with having too large of a learning rate in gradient descent?