DSC 140A - Homework 01

Due: Wednesday, April 10

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 PM.

Problem 1.

In this problem, we'll show that the gradient of $f(\vec{x}) = \vec{x}^T A \vec{x}$ is $2A\vec{x}$, where A is a symmetric $n \times n$ matrix and $\vec{x} \in \mathbb{R}^n$. This is a useful result, but it's also a good exercise for reviewing topics in matrix-vector algebra and multivariate calculus.

a) To compute the gradient, we need to compute $\partial f / \partial x_1$, $\partial f / \partial x_2$, and so on. To do this, we'll start by expanding $\vec{x}^T A \vec{x}$ until we see the coordinates of $\vec{x}$.

Let the entries of A be

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}.$$

Show that $f(\vec{x}) = \vec{x}^T A \vec{x} = \sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j$.

b) Show that:

$$\frac{\partial f}{\partial x_k} \left(\sum_{i=1}^n \sum_{j=1}^n a_{ij} x_i x_j \right) = 2 \sum_{j=1}^n a_{kj} x_j$$

In other words, show that $\partial f / \partial x_1$ is $2 \sum_{j=1}^n a_{1j} x_j$, $\partial f / \partial x_2$ is $2 \sum_{j=1}^n a_{2j} x_j$, and so on.

c) The gradient vector is the vector of partial derivatives:

$$(\partial f / \partial x_1, \partial f / \partial x_2, \ldots, \partial f / \partial x_n)^T$$

We have found so far that the gradient is:

$$\begin{pmatrix} 2 \sum_{j=1} a_{1j} x_j \\ 2 \sum_{j=1} a_{2j} x_j \\ \vdots \\ 2 \sum_{j=1} a_{nj} x_j \end{pmatrix}$$

Show that this is equal to $2A\vec{x}$.

Hint: You can work backwards, expanding $2A\vec{x}$ and showing that it equals the gradient.

Solution: To compute the gradient, we follow the general strategy outlined in lecture of:

1. Expand until we see the components of $\vec{x}$.
2. Compute the gradient vector of partial derivatives: $(\partial f / \partial x_1, \partial f / \partial x_2, \ldots)^T$
3. Try to rewrite the gradient in vector form.

Let the entries of A be

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}.$$

In discussion, it was shown that

$$\vec{x}^T A\vec{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_ix_j.$$

It may help to visualize all of these terms in a table:

	$j=1$	$j=2$	$\cdots$	$j=n$
$i=1$	$a_{11}x_1^2$	$a_{12}x_1x_2$	$\cdots$	$a_{1n}x_1x_n$
$i=2$	$a_{21}x_2x_1$	$a_{22}x_2^2$	$\cdots$	$a_{2n}x_2x_n$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$i=n$	$a_{n1}x_nx_1$	$a_{n2}x_nx_2$	$\cdots$	$a_{nn}x_n^2$

This table contains all of the terms in the double summation. If we added up all of the terms in this table, we would get exactly $\vec{x}^T A\vec{x}$.

Next, we compute the gradient vector, starting with $\partial f/\partial x_1$. We have:

$$\begin{aligned} \partial f/\partial x_1 &= \frac{\partial}{\partial x_1}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_ix_j \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \frac{\partial}{\partial x_1} a_{ij}x_ix_j \end{aligned}$$

If neither i or j are equal to one, the derivative will be zero. Which are not zero, and what are they? This is where the table comes in handy. The partial of each entry with respect to x_1 is:

	$j=1$	$j=2$	$\cdots$	$j=n$
$i=1$	$2a_{11}x_1$	$a_{12}x_2$	$\cdots$	$a_{1n}x_n$
$i=2$	$a_{21}x_2$	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$i=n$	$a_{n1}x_n$	0	$\cdots$	0

Since A is symmetric, $a_{12} = a_{21}$, $a_{13} = a_{31}$, etc. Gathering like terms, we find:

$$\begin{aligned} &= 2a_{11}x_1 + 2a_{12}x_2 + \ldots + 2a_{1n}x_n \\ &= 2\sum_{j=1}^{n} a_{1j}x_j \end{aligned}$$

We can guess that, in general, $\partial f/\partial x_i = 2\sum_{j=1}^{n} a_{ij}x_j$. Therefore, the gradient vector is:

$$\frac{df}{d\vec{x}} = \begin{pmatrix} 2\sum_{j=1} a_{1j}x_j \\ 2\sum_{j=1} a_{2j}x_j \\ \vdots \\ 2\sum_{j=1} a_{nj}x_j \end{pmatrix}$$

To show that this equals $2A\vec{x}$, we can work backwards by expanding $2A\vec{x}$. We have:

$$\begin{aligned}
2A\vec{x} &= 2\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \\
&= 2\begin{pmatrix} \sum_{j=1}^n a_{1j}x_j \\ \sum_{j=1}^n a_{2j}x_j \\ \vdots \\ \sum_{j=1}^n a_{nj}x_j \end{pmatrix} \\
&= \begin{pmatrix} 2\sum_{j=1}^n a_{1j}x_j \\ 2\sum_{j=1}^n a_{2j}x_j \\ \vdots \\ 2\sum_{j=1}^n a_{nj}x_j \end{pmatrix} \\
&= \frac{df}{d\vec{x}}
\end{aligned}$$

Problem 2.

The following is a common question you might see in an interview for a machine learning job. The question is: Should you scale your features before performing least squares?

Your friend thinks that scaling the features before performing least squares regression is a good idea. Their logic is that scaling sometimes improves the performance of k-nearest neighbors predictors, so it might help with least squares as well. But you're not so sure.

In this problem, we'll show that scaling the features before performing least squares regression does not actually change the predictions that are made, and so it doesn't improve the performance of the model[1].

Hint: It might be helpful to use some of the matrix-vector algebra properties covered in discussion. If you use a property that wasn't listed in discussion or lecture, it's OK, but make sure to explicitly state the property you're using.

a) Let $(\vec{x}^{(1)}, y_1), (\vec{x}^{(2)}, y_2), \ldots, (\vec{x}^{(n)}, y_n)$ be a training set of n feature vectors in $\mathbb{R}^d$ and their corresponding labels. Recall that the decision matrix X is

$$X = \begin{pmatrix} 1 & \vec{x}_1^{(1)} & \vec{x}_2^{(1)} & \cdots & \vec{x}_d^{(1)} \\ 1 & \vec{x}_1^{(2)} & \vec{x}_2^{(2)} & \cdots & \vec{x}_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \vec{x}_1^{(n)} & \vec{x}_2^{(n)} & \cdots & \vec{x}_d^{(n)} \end{pmatrix}$$

Suppose we will scale feature i by a constant factor of c_i in each feature vector. The result is a scaled data set whose design matrix X_C is:

$$X_C = \begin{pmatrix} 1 & c_1\vec{x}_1^{(1)} & c_2\vec{x}_2^{(1)} & \cdots & c_d\vec{x}_d^{(1)} \\ 1 & c_1\vec{x}_1^{(2)} & c_2\vec{x}_2^{(2)} & \cdots & c_d\vec{x}_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & c_1\vec{x}_1^{(n)} & c_2\vec{x}_2^{(n)} & \cdots & c_d\vec{x}_d^{(n)} \end{pmatrix}$$

Find a diagonal matrix C so that $X_C = XC$.

[1] This is not to say that scaling is never useful when doing least squares regression. Scaling the features can make the weights easier to interpret, for example, or help with numerical stability.

Solution: We can find what the diagonal matrix C should be by writing out the matrix multiplication XC and comparing it to X_C.

First, how big should C be? Since X is $n \times (d+1)$, we need C to be $(d+1) \times (d+1)$ so that XC is also $n \times (d+1)$.

Let's say that C is the diagonal matrix:

$$C = \begin{pmatrix} a_0 & 0 & 0 & \cdots & 0 \\ 0 & a_1 & 0 & \cdots & 0 \\ 0 & 0 & a_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_d \end{pmatrix}$$

We're trying to find out what the a_i should be.

When we multiply XC, we get:

$$\begin{aligned} XC &= \begin{pmatrix} 1 & \vec{x}_1^{(1)} & \vec{x}_2^{(1)} & \cdots & \vec{x}_d^{(1)} \\ 1 & \vec{x}_1^{(2)} & \vec{x}_2^{(2)} & \cdots & \vec{x}_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \vec{x}_1^{(n)} & \vec{x}_2^{(n)} & \cdots & \vec{x}_d^{(n)} \end{pmatrix} \begin{pmatrix} a_0 & 0 & 0 & \cdots & 0 \\ 0 & a_1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_d \end{pmatrix} \\ &= \begin{pmatrix} a_0 + a_1\vec{x}_1^{(1)} + a_2\vec{x}_2^{(1)} + \cdots + a_d\vec{x}_d^{(1)} \\ a_0 + a_1\vec{x}_1^{(2)} + a_2\vec{x}_2^{(2)} + \cdots + a_d\vec{x}_d^{(2)} \\ \vdots \\ a_0 + a_1\vec{x}_1^{(n)} + a_2\vec{x}_2^{(n)} + \cdots + a_d\vec{x}_d^{(n)} \end{pmatrix} \end{aligned}$$

Since we want this to equal X_C, we compare term by term and find that we should set:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= c_1 \\ a_2 &= c_2 \\ &\vdots \\ a_d &= c_d \end{aligned}$$

Therefore, the matrix C is:

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & c_1 & 0 & \cdots & 0 \\ 0 & 0 & c_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & c_d \end{pmatrix}$$

b) For the unscaled data, the optimal parameter vector is given by $\vec{w}^* = (X^T X)^{-1} X^T \vec{y}$. When we scale the features, however, the optimal weight vector becomes $\vec{w}_C^* = (X_C^T X_C)^{-1} X_C^T \vec{y}$.

Show that $\vec{w}_C^* = C^{-1}\vec{w}^*$ by using the fact that $X_C = XC$.

Hint: $(AB)^{-1} = B^{-1}A^{-1}$ (as long as A and B are invertible).

Solution: We know that $X_C = XC$, so we can write:

$$\begin{aligned}\vec{w}_C^* &= (X_C^T X_C)^{-1} X_C^T \vec{y} \\ &= (CX^TXC)^{-1} CX^T \vec{y}\end{aligned}$$

Now we can use the fact that $(AB)^{-1} = B^{-1}A^{-1}$, as long as A and B are invertible, but we have to be careful! X^T and X are *not* invertible (they aren't square matrices), but X^TX is square (and, presumably, invertible). C is also invertible. So, iteratively applying the fact, we get $(CX^TXC)^{-1} = C^{-1}(X^TX)^{-1}\left(C^T\right)^{-1}$, and we get:

$$= C^{-1}(X^TX)^{-1}\left(C^T\right)^{-1} CX^T\vec{y}$$

Since C is diagonal, $C^T = C$, so we can simplify:

$$\begin{aligned}&= C^{-1}(X^TX)^{-1}C^{-1}CX^T\vec{y} \\ &= C^{-1}\underbrace{(X^TX)^{-1}X^T\vec{y}}_{\vec{w}^*} \\ &= C^{-1}\vec{w}^*\end{aligned}$$

c) Consider a new point $\vec{x}$ for which we want to make a prediction.

The prediction made by the original model is $\vec{w}^* \cdot \operatorname{Aug}(\vec{x})$.

The prediction made by the new model trained on scaled data is $\vec{w}_C^* \cdot \operatorname{Aug}(\vec{x}_C)$, where $\vec{x}_C$ is $\vec{x}$ scaled by the same factors as the training data. It can be shown that $\operatorname{Aug}(\vec{x}_C) = C\operatorname{Aug}(\vec{x})$.

Using these facts and the result of the previous parts, show that $\vec{w}^* \cdot \operatorname{Aug}(\vec{x}) = \vec{w}_C^* \cdot \operatorname{Aug}(\vec{x}_C)$; that is, both models make exactly the same prediction, and scaling had no effect.

Hint: The inverse of a diagonal matrix is also diagonal. The transpose of a diagonal matrix is the same as the original matrix.

Solution: Since $w_C^* = C^{-1}w^*$ and $\operatorname{Aug}(\vec{x}_C) = C\operatorname{Aug}(\vec{x})$, we have:

$$\vec{w}_C^* \cdot \operatorname{Aug}(\vec{x}_C) = C^{-1}\vec{w}^* \cdot C\operatorname{Aug}(\vec{x})$$

We want to somehow "cancel" the C and C^{-1} terms, which means getting them next to one another. For this, we can use the fact that $\vec{a}\cdot\vec{b} = \vec{a}^T\vec{b}$ for vectors $\vec{a}$ and $\vec{b}$ to write:

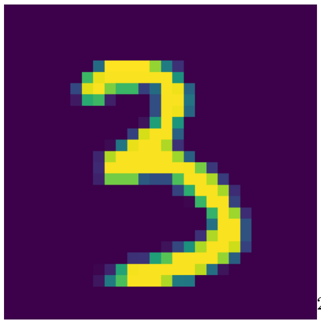
$$\begin{aligned}&= \left(C^{-1}\vec{w}^*\right)^T(C\operatorname{Aug}(\vec{x})) \\ &= (\vec{w}^*)^T(C^{-1})^T C\operatorname{Aug}(\vec{x})\end{aligned}$$

Because C is diagonal, $(C^{-1})^T = (C^T)^{-1} = C^{-1}$, so we can simplify:

$$\begin{aligned}&= (\vec{w}^*)^T C^{-1}C\operatorname{Aug}(\vec{x}) \\ &= (\vec{w}^*)^T\operatorname{Aug}(\vec{x}) \\ &= \vec{w}^*\cdot\operatorname{Aug}(\vec{x})\end{aligned}$$

Problem 3.

In this problem, you'll implement a k-nearest neighbor classifier and use it to predict whether an image of a handwritten digit (like the one shown below) is either a 3 or a 7.

[2]

Along the way, you'll see some of the common technical issues that might come with using a machine learning algorithm and how to fix them.

The file linked below is a subset of the MNIST dataset, which contains images of handwritten digits. It is provided in `npz` format, which is a compressed file format that can store multiple arrays; it can be read using the `numpy` library's `np.load` function.

`https://f000.backblazeb2.com/file/jeldridge-data/014-mnist37/mnist.npz`

When loaded it behaves like a Python dictionary with the following keys:

- `train`: A 784 by 12,396 array containing the training images. Each column represents one training image that has been "flattened" into a 784-dimensional vector (784 dimensional because the original images are 28 by 28 pixels, and 28 times 28 is 784).
- `train_labels`: An array of 12,396 entries containing the labels for the training images. Each entry is either a 3 or a 7.
- `test`: A 784 by 2,038 array containing the test images. Each column represents one test image that has been flattened into a 784-dimensional vector.
- `test_labels`: An array of 2,038 entries containing the labels for the test images. Each entry is either a 3 or a 7.

In lecture, code was given for a function `knn_predict` that implemented a k-nearest neighbor classifier. You're encouraged to use that code for this problem, but if you simply plug in the training and test data, you'll find that the classifier doesn't work as expected. There isn't a bug in the code; rather, the issue is that the input is not in a form that the code expects. In that sense, this problem is meant to be practice for how to manage these small technical issues (since you will likely encounter them at some point, either in this class or in your career).

Some errors you might get are:

- "`ValueError:` operands could not be broadcast together with shapes ...": this error occurs when you try to add or subtract two arrays of different shapes. Check: does the `knn_train` function expect the training data to be in a $n \times d$ matrix, or a $d \times n$ matrix? What are you providing it with?
- "`TypeError:` Cannot cast array data from dtype('float64') to dtype('int64')...": This error is probably being raised by `np.bincount`. It wants an array of integers. Is that what you're giving it? You might want to use the `.astype()` method to convert an array to a different data type.

After fixing these errors, your code should run – but we're not done just yet. Compute the accuracy of your classifier on the first 100 test examples using $k = 1$ neighbors. You will likely see something like 81% accuracy. This might sound good, but it's actually much less than it should be.

[2]If you'd like to plot the handwritten digits (for fun), you can use `plt.imshow(x.reshape(28, 28))` from the `matplotlib` library, where `x` is a 784-dimensional vector.

The issue is that the training images are all encoded as 8-bit "unsigned integers" (or `uint8`, in `numpy`'s language), and an 8-bit unsigned integer can only store values from 0 to 255. If you subtract two 8 bit integers (as we do when we compute the distance between two images), the result might be a negative number *mathematically*, but the computer will not be able to store the result as a negative number in an 8-bit unsigned integer. Instead, it will "wrap around" and store the result as a positive number. This means that the distance between two images is not what you'd expect. For example, if you compute:

```
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> y = np.array([4, 5, 6], dtype=np.uint8)
>>> x - y
```

you expect to get `array([-3, -3, -3])`, but you actually get `array([253, 253, 253], dtype=uint8)`.

To fix this, you should convert the training and test images to a data type that can represent negative numbers as large as needed. `np.int16` should be enough, since it can represent numbers from $-32{,}768$ to 32,767.

After fixing all of these, you should have a classifier that gets very close to 100% accuracy on the first 100 test examples. *Hint*: you should not need to change the code of `knn_predict` at all; you just need to change its inputs.

What you should turn in for this problem:

- the final accuracy of the $k = 1$ nearest neighbor classifier on the first 100 test examples after fixing the data type issue.
- The predictions your classifier makes on the 0^{th}, 500^{th}, and 1000^{th} test examples. E.g.,: 7, 7, 7 if the classifier predicts 7 for all three examples (which it shouldn't!)
- A short list of the changes you had to make to get these results.
- Your code, either as a screenshot or pasted into the document.

Solution: There are several things we need to do with the data to get the expected result.

First, we need to transpose the training and test data. The `knn_train` function wants a parameter, `X_train`, containing all of the training data. There are two conventions about how to store the data in an array: either each row is a data point, or each column is a data point. The `knn_train` function expects the former, but the data is actually stored in the latter format. We can fix this by transposing the data. Assuming we've already loaded the data with: `data = np.load('mnist.npz')`, we can do this with `X_train = data['train'].T`.

We also need to note that the `knn_predict` function expects an argument, `x`, which is a single test data point to make a prediction for. If you try passing in the entire test data set, you'll get an error. Instead, you should loop over the test data and make predictions one by one.

Next, we need to fix the data type. `np.uint8` cannot represent negative numbers, so we should convert the data to a signed type, like `np.int16`. We can do this with `data['train'].astype(np.int16)`.

After this, your predictions on the 0^{th}, 500^{th}, and 1000^{th} test examples should all be "3", and you should achieve exactly 98% accuracy on the first 100 test examples.

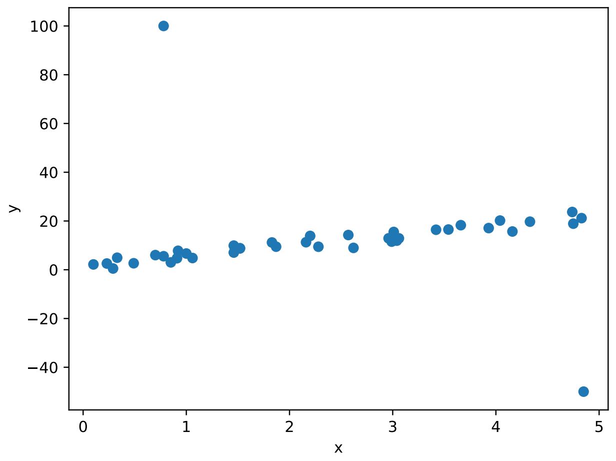
Problem 4.

The file at the link below contains a simple data set suitable for regression.

`https://f000.backblazeb2.com/file/jeldridge-data/002-regression_outlier/data.csv`

The first column contains the x values (the predictor variable) and the second column contais the y values (the target).

The plot below shows the data:



Notably, the data contains outliers which may affect our regression.

Fit a linear function of the form $w_0 + w_1 x$ by minimizing the mean squared error. Report w_0 and w_1, and include your code.

Do not use `sklearn`, `scipy`, or any library except for `numpy` and `matplotlib` for this problem. You may use the code given in lecture for least squares regression.

Solution:

```
# load the data
X, y = np.loadtxt('./data.csv', delimiter=',').T

# make an augmented design matrix by adding a column of ones
# to X
X_aug = np.column_stack((
    np.ones_like(X), X
))

# fit by minimizing squared error
w_sq = np.linalg.lstsq(X_aug, y)[0]
```

We find, approximately: $w_0 \approx 11.26$ and $w_1 \approx 0.18$.