Question 1. Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, “HW[n] Write-Up”
2. Submit all code needed to reproduce your results, “HW[n] Code”.
3. Submit your test set evaluation results, “HW[n] Test Set”.

After you’ve submitted your homework, be sure to watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked with Weiran Liu and Katherine Li. You know there is a certain time of the year when you don’t want to do anything. That time is now. There are certain types of things that come at you and you cannot avoid them. Those things are homeworks.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student’s solutions. I have credited all external sources in this write up.

I certify that all solutions are entirely in my words and that I have not looked at another student’s solutions. I have credited all external sources in this write up.

Signature: [illegible]

Question 2. K-SVD

As you have seen in earlier homework problems, sparse representations are powerful. When we know the right dictionary of features within which our input is likely to have a sparse representations (possibly with some additional small non-sparse noise), we can use techniques like the LASSO or OMP (or even just rounding/thresholding) to recover the coefficients. As we have seen, this has a fundamentally better bias/variance trade-off.

But what if we don't know the dictionary? How can the dictionary itself be found from the training data? This is known as *dictionary learning.* In this problem, we will introduce the K-SVD algorithm (Aharon, M., Elad, M. and Bruckstein, A., 2006. "k-SVD: An algorithm for designing overcomplete dictionaries for sparse representation." IEEE Transactions on signal processing, 54(11), pp.4311-4322.) for unsupervised dictionary learning. (The naive perspective on directly supervised dictionary learning would reveal the dictionary to us, so that is not very interesting. However, it should be clear that the dictionary learning algorithm here can be adapted to deal with the case where we already know some dictionary elements to start with and want to learn more. It should also be clear that there are EM-based approaches to do dictionary learning in a more fully Bayesian perspective, as well as that various deep neural-network architectures are in effect trying to do dictionary learning at early layers to find the features that combine to best represent the signals being presented.) The K-SVD setting is where we would like to find a dictionary that enables good sparse fits to our data in the standard least-squares sense.

Mathematically, given an input matrix $\mathbf{X} \in \mathbb{R}^{n\times d}$, where n is the number of data points and d is the dimension of each data point, we want to solve the following optimization problem:

$$\begin{aligned} \underset{\mathbf{D}\in\mathbb{R}^{K\times d},\mathbf{Z}\in\mathbb{R}^{n\times K}}{\text{minimize}} \quad & \|\mathbf{X}-\mathbf{Z}\mathbf{D}\|_{\mathrm{F}}^2 \\ \text{subject to} \quad & \|\vec{z}_i^{\top}\|_0 \leq s \text{ for all } i. \end{aligned} \tag{1}$$

We explain the notation a bit more here:

- The matrix $\mathbf{D} \in \mathbb{R}^{K\times d}$ is called a *dictionary* or a *codebook.* Each row $\vec{D}_k$ represents a dictionary vector/element/codeword/atom.
- Each data sample $\vec{x}_i$ is presumed to be a combination of the dictionary vectors and the coefficients are given by $\vec{z}_i^T$, the ith row of $\mathbf{Z}$. See equation (2).
- We denote by $\|\vec{v}\|_0$ the "zeroth-norm" of vector $\vec{v}$, which is defined to be the number of nonzero entries in the vector $\vec{v}$.

To summarize, we have $\mathbf{X} \in \mathbb{R}^{n\times d}$. $\mathbf{Z} \in \mathbb{R}^{n\times K}$, and $\mathbf{D} \in \mathbb{R}^{K\times d}$ and that

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1^{\top} \\ \vec{x}_2^{\top} \\ \vdots \\ \vec{x}_n^{\top} \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} \vec{D}_1^{T} \\ \vec{D}_2^{T} \\ \vdots \\ \vec{D}_K^{T} \end{bmatrix} \quad \text{and} \quad \mathbf{Z} = \begin{bmatrix} \vec{z}_1^{T} \\ \vec{z}_2^{T} \\ \vdots \\ \vec{z}_n^{T} \end{bmatrix} = \left[\vec{Z}_1, \vec{Z}_2, \ldots, \vec{Z}_K\right]$$

with $\vec{D}_k \in \mathbb{R}^d$, $\vec{z}_i \in \mathbb{R}^K$ and $\vec{Z}_k \in \mathbb{R}^n$.

And to summarize the goal: Our goal is to find a dictionary $\mathbf{D}$ such that we can approximate the data samples $\vec{x}_i$ by a sparse linear combination of atoms in $\mathbf{D}$:

$$\vec{x}_i^{\top} \approx \sum_{k=1}^{K} z_{ik}\vec{D}_k^{\top}, \tag{2}$$

where the coefficients of linear combination are given in $\mathbf{Z}$ (which also needs to be determined) and should have at most s nonzero entries for each sample.

A greedy iterative algorithm called K-SVD is proposed for the above optimization problem. The algorithm optimizes the objective over $\mathbf{D}$ and $\mathbf{Z}$ in an alternating fashion. The pseudo code of the algorithm is given in Algorithm 1. It alternates between two stages: Sparse coding (Algorithm 2) and Update Codebook (Algorithm 4). The Sparse Coding procedure finds a sparse representation for each data sample with Algorithm 3 based on a given dictionary. The procedure for updating the dictionary updates each row of the dictionary with Algorithm 5 in a for loop, while modifying the effected representation coefficients correspondingly.

We have two goals in this problem: (1) to establish a relationship of K-SVD to the K-means algorithm you already know, and (2) to show that this K-SVD algorithm converges to a local minimum. (For this reason in practice, K-SVD is run with multiple random initial conditions to make sure that a good minimum is found. Just like we have seen for K-means in earlier homeworks.)

Algorithm 1: K-SVD

```
Input: Data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$; Number of atoms $K$; Sparsity constraint $s$
Output: A dictionary $\mathbf{D} \in \mathbb{R}^{K \times d}$, and the coefficient matrix $\mathbf{Z} \in \mathbb{R}^{n \times K}$
function K-SVD($\mathbf{X}, K, s$):
    $I$nitialize $\mathbf{D} \leftarrow$ randomly chosen $K$ rows of the dataset $\mathbf{X}$ (without replacement)
    $I$nitialize $\mathbf{Z} \leftarrow$ Sparse-coding($\mathbf{D}, \mathbf{0}, \mathbf{X}, s$)
    while not converged do
        $\mathbf{Z} \leftarrow$ Sparse-coding($\mathbf{D}, \mathbf{Z}, \mathbf{X}, s$)
        Update-dictionary($\mathbf{D}, \mathbf{Z}, \mathbf{X}$)
    end
    return $\mathbf{D}, \mathbf{Z}$
end function
```

Algorithm 2: Sparse-coding

```
Input: Dictionary $\mathbf{D}$; Coefficient matrix $\mathbf{Z} \in \mathbb{R}^{n \times K}$; Data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$; Sparsity constraint $s$
Output: The coefficient matrix $\mathbf{Z} \in \mathbb{R}^{n \times K}$
function Sparse-coding($\mathbf{D}, \mathbf{Z}, \mathbf{X}, \mathrm{s}$):
    for $i = 1, \dots, n$ do
        $\vec{z}_i' =$ Sparse-coding-single($\mathbf{D}, \vec{x}_i, s$)
        if $\|\vec{x}_i^\top - \vec{z}_i^\top \mathbf{D}\|_2 > \|\vec{x}_i^\top - (\vec{z}_i')^\top \mathbf{D}\|_2$ then
            $\vec{z}_i \leftarrow \vec{z}_i'$
        end
    end
    return $\mathbf{Z} = \begin{bmatrix} \vec{z}_1^\top \\ \vdots \\ \vec{z}_n^\top \end{bmatrix}$
end function
```

(a) (Relationship to K-means) Recall the set-up of K-means: Given a data matrix $\mathbf{X}$, K-means algorithm *partitions* the data into K disjoint groups $\vec{\pi} = \{\pi_1, \pi_2, \dots, \pi_K\}$. Each sample $\vec{x}_i$ is contained in *exactly one partition* π_j. Recall that the K-means algorithm tries to solve the following optimization problem:

$$\underset{\vec{\pi}, \mu \in \mathbb{R}^{K \times d}}{\text{minimize}} \quad \sum_{k=1}^{K} \sum_{i \in \pi_k} \|\vec{x}_i - \vec{\mu}_k\|_2^2. \tag{3}$$

Algorithm 3: Sparse-coding-single

```
Input: Dictionary $\mathbf{D}$; Data sample $\vec{x} \in \mathbb{R}^d$; Sparsity constraint $s$
Output: A coefficient vector $\vec{z} \in \mathbb{R}^K$
1  function Sparse-coding-single($\mathbf{D}, \vec{x}, s$):
2      $I$nitialize $\vec{z} \leftarrow \vec{0} \in \mathbb{R}^k$
3      $I$nitialize the basis $\mathcal{B}_0 = \{\}$
4      for $j = 1, \ldots, s$ do
5          Find the index of the best dictionary vector $\vec{D}_{k^{(j)}} \in \mathbb{R}^d$ by solving:
               $\hat{\beta}^{(j)}, k^{(j)} = \arg \min_{\vec{\beta} \in \mathbb{R}^j} \min_{k \in [K]} \|\vec{x} - \sum_{\vec{D}_l \in \mathcal{B}_{j-1} \cup \{\vec{D}_k\}} \beta_l \vec{D}_l\|_2^2$
6          Update $\mathcal{B}_j \leftarrow \mathcal{B}_{j-1} \cup \{\vec{D}_{k^{(j)}}\}$
7      end
8      for $\vec{D}_l \in \mathcal{B}_s$ do
9          $\vec{z}_l \leftarrow \beta_l^{(s)}$
10     end
11     return $\vec{z}$
12 end function
```

Algorithm 4: Update-dictionary

```
Input: Dictionary $\mathbf{D}$; Coefficient matrix $\mathbf{Z} \in \mathbb{R}^{n \times K}$; Data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$
Output: Dictionary $\mathbf{D} \in \mathbb{R}^{K \times d}$
1  function Update-dictionary($\mathbf{D}, \mathbf{Z}, \mathbf{X}$):
2      for $k = 1, \ldots, K$ do
3          Update $k$-th row $\vec{D}_k^\top \leftarrow$ Update-dictionary-single($\mathbf{D}, \mathbf{Z}, \mathbf{X}, k$)
4      end
5      return $\mathbf{D} = \begin{bmatrix} \vec{D}_1^\top \\ \vdots \\ \vec{D}_K^\top \end{bmatrix}$.
6  end function
```

Algorithm 5: Update-dictionary-single

```
**Input:** Dictionary $\mathbf{D}$; Coefficient matrix $\mathbf{Z} \in \mathbb{R}^{n \times K}$; Data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$; Index k
**Output:** Return a row vector $\vec{\tilde{d}} \in \mathbb{R}^{1 \times d}$
function Update-dictionary-single($\mathbf{D}, \mathbf{Z}, \mathbf{X}, k$):
    for $k = 1, \ldots, K$ do
        Compute the error matrix $\mathbf{E}_k \in \mathbb{R}^{n \times d}$ that represent how well things are represented without the
          dictionary vector $\vec{D}_k^\top$:
              $$\mathbf{E}_k = \mathbf{X} - \sum_{j \neq k} \vec{z}_j \vec{D}_j^\top$$
        Find the indices of data samples whose sparse representation uses the dictionary vector $\vec{D}_k^\top$.
        In other words, find the indices corresponding to non-zero entries in the coefficient vector $\vec{z}_k$
              $$\omega_k = \{i | i \in [n], \vec{z}_{k,i} \neq 0\}.$$
          and let us denote $\omega_k = \{i_1, \ldots, i_{|\omega_k|}\}$.
        Find the matrix $\mathbf{E}_k^{\omega_k} \in \mathbb{R}^{|\omega_k| \times d}$ by choosing those rows of $\mathbf{E}_k$ whose index belongs to the set $\omega_k$, i.e.,
          matrix $\mathbf{E}_k^{\omega_k}$ is computed by picking the rows with indices $i_1, \ldots, i_{|\omega_k|}$.
        Compute the SVD of $\mathbf{E}_k^{\omega_k}$:
              $$\mathbf{E}_k^{\omega_k} = \mathbf{U} \mathbf{\Lambda} \mathbf{V}^\top$$
          where $\mathbf{U} \in \mathbb{R}^{|\omega_k| \times |\omega_k|}$, $\mathbf{\Lambda} \in \mathbb{R}^{|\omega_k| \times d}$ and $\mathbf{V} \in \mathbb{R}^{d \times d}$. We assume $\mathbf{\Lambda}$ is a diagonal matrix with
          non-decreasing diagonal entries. Let $U_1 \in \mathbb{R}^{|\omega_k|}$ and $V_1 \in \mathbb{R}^d$ denote the first column of the matrices $\mathbf{U}$
          and $\mathbf{V}$ respectively.
        Update $\vec{D}_k \leftarrow V_1$
        Update only the non-zero entries of the vector $\vec{z}_k^\top$:
        for $j = 1, \ldots, |\omega_k|$ do
            $\vec{z}_{k,i_j} = U_{1,i_j} \Lambda_{11}$
        end
    end
    return $\vec{D}_k^\top$.
end function
```

Let us connect the K-means optimization problem (3) with that given by equation (1). Let μ denote a $K \times d$ matrix and $\tilde{\mathbf{Z}}$ denote an $n \times K$ matrix which are given by

$$\mu = \begin{bmatrix} \vec{\mu}_1^\top \\ \vdots \\ \vec{\mu}_K^\top \end{bmatrix} \text{ and } \quad \tilde{\mathbf{Z}} = \begin{bmatrix} \vec{\tilde{z}}_1^\top \\ \vdots \\ \vec{\tilde{z}}_n^\top \end{bmatrix}.$$

Show that the objective function given in the problem (3) is equivalent to

$$\begin{aligned} &\underset{\tilde{\mathbf{Z}} \in \mathbb{R}^{n \times K}, \mu \in \mathbb{R}^{K \times d},}{\text{minimize}} && \|\mathbf{X} - \tilde{\mathbf{Z}}\mu\|_{\mathrm{F}}^2 \\ &\text{subject to} && \vec{\tilde{z}}_i \in \{0,1\}^K \text{ and } \|\vec{\tilde{z}}_i\|_0 = 1 \text{ for all } i = 1, \dots, n. \end{aligned} \tag{4}$$

Conclude that K-means is equivalent to K-SVD with $s = 1$ with an additional constraint of forcing the coefficient entries in the matrix $\tilde{\mathbf{Z}}$ to take values in the set $\{0, 1\}$.

$$\begin{aligned} &\min \|\mathbf{X} - \tilde{\mathbf{Z}}\mu\|_{\mathrm{F}}^2 \\ =&\min \sum_{i=1}^{n} \|\vec{x}_i - \vec{\mu}^\top \vec{\tilde{z}}_i\|_2^2 \\ &\text{Because } \vec{\tilde{z}}_i \text{ only has one non-zero entry} \\ =&\min \sum_{i=1}^{n} \|\vec{x}_i - \vec{\mu}_{k(i)}\|_2^2 \\ &\text{The expression just means to minimize the MSE the data point} \\ &\text{to its corresponding class centroid} \\ =&\min \sum_{k=1}^{K} \sum_{i \in \pi_k} \|\vec{x}_i - \vec{\mu}_k\|_2^2 \end{aligned}$$

Because two equations are equivalent, we can conclude that K-means is equivalent to K-SVD with $s = 1$ with an additional constraint of forcing the coefficient entries in the matrix $\tilde{\mathbf{Z}}$ to take values in the set $\{0, 1\}$.

(b) **Show that at each step $j = 1, 2, \ldots, s$ in Algorithm 3 for finding a sparse solution, the min value of the objective ($\min_{\vec{\beta}\in\mathbb{R}^j} \min_{k\in[K]} \|\vec{x} - \sum_{\vec{D}_l \in \mathcal{B}_{j-1}\cup\{\vec{D}_k\}} \beta_l \vec{D}_l\|_2^2$) is non-increasing.**

The following hint may or may not be useful for you but is worth thinking through. *Hint: Can you identify this algorithm as OMP?*

Yes. I can see it's similar to OMP because at each iteration, we add a new dimension to the existing set and try to find the solution that gives the minimum residual. At each iteration we always keep the previously selected variables the same and find a smaller residue error. Here is what we want:

$$\begin{aligned}
&\|\vec{x} - \sum_{\vec{D}_l \in \mathcal{B}_{j-1}\cup\{\vec{D}_{k(j)}\}} \beta_l \vec{D}_l\|_2^2 \\
=&\|\vec{x} - \sum_{\vec{D}_l \in \mathcal{B}_{j-1}} \hat{\beta}_l \vec{D}_l - \beta_j \vec{D}_{k(j)}\|_2^2 \\
=&\|\vec{x} - \sum_{\vec{D}_l \in \mathcal{B}_{j-1}} \hat{\beta}_l \vec{D}_l\|_2^2 + \|\beta_j \vec{D}_{k(j)}\|_2^2 \\
\leq&\|\vec{x} - \sum_{\vec{D}_l \in \mathcal{B}_{j-1}} \hat{\beta}_l \vec{D}_l\|_2^2
\end{aligned}$$

(c) (Sparse coding decreases the objective) **Conclude from the above part that Algorithm 2 cannot increase the value of the objective function (1).** (In practice, people use straight matching pursuit — Algorithm 6 — to replace Algorithm 3, which is much more efficient, but gets comparable performance in most practical settings where there isn't a huge dynamic range of coefficients and the target sparsity s is low. But showing that matching pursuit usually works is omitted here.)

Algorithm 6: Matching-pursuit

```
Input: Dictionary $\mathbf{D}$; Data sample $\vec{x} \in \mathbb{R}^d$; Sparsity constraint $s$
Output: A coefficient vector $\vec{z} \in \mathbb{R}^K$.
function Sparse-coding-single-in-practice($\mathbf{D}, \vec{x}, s$):
    $I$nitialize $\vec{z} \leftarrow \vec{0} \in \mathbb{R}^k$
    $I$nitialize the basis $\mathcal{B}_0 = \{\}$
    $I$nitialize the residue $\vec{r}_0 = \vec{x}$
    for $j = 1, \dots, s$ do
        Find the best dictionary vector $\vec{D}_{k^{(j)}}^\top \in \mathbb{R}^d$ by solving:
            $k^{(j)} = \arg\max_{k\in[K]} \frac{|\vec{r}_{j-1}^\top \vec{D}_k|}{\|\vec{D}_k\|_2} = \arg\min_{k\in[K]} \min_{\beta_k\in\mathbb{R}} \left\|\vec{r}_{j-1} - \beta_k \vec{D}_k\right\|_2^2$
        Update the coefficient $\vec{z}_{k^{(j)}} \leftarrow \frac{|r_{j-1}^\top \vec{D}_k}{\|\vec{D}_k\|_2^2}$
        Update the residue $\vec{r}_j \leftarrow \vec{r}_{j-1} - \vec{z}_{k^{(j)}} \vec{D}_{k^{(j)}}$
    end
    return $\vec{z}$.
end function
```

Use the conclusion from the previous part we know that algorithm 2 cannot increase the value of the objective function (1) because it loops through the data points and find a smaller residue error after each iteration.

(d) **Show that the single-atom dictionary update given by Algorithm 5 does not increase the objective function while preserving the sparsity constraint.** (You may or may not find the following hint useful.)

Hint: Recall the Eckart-Young theorem.

$$
\begin{aligned}
&\|X - ZD\|_F^2\\
=&\|E_k - \vec{z}_k D_k^\top\|_F^2\\
&\text{picking the rows with indices } i_1, \ldots, i_{|\omega_k|} \text{ to get } E_k^{w_k}\\
=&\|E_k^{w_k} - \vec{z}_k D_k^\top\|_F^2 + \|E_k^{C(w_k)}\|_F^2\\
&\text{we set } D_k \text{ to be } V_1 \text{ and the nonzero elements of } \vec{z}_k \text{ to be } U_1\Sigma_{11}\\
=&\|E_k^{w_k} - U_1\Sigma_{11}V_1\|_F^2\\
&\text{By Eckart-Young theorem, this gives the smallest reconstruction error}
\end{aligned}
$$

(e) (Updating dictionary decreases the objective) **Conclude that the iterations of the K-SVD algorithm put together cannot increase the objective function and hence the objective function must converge.**

Again, we use the conclusion from the previous part to show that K-SVD algorithm loops through the atoms, and it decreases the error which is guaranteed by the Eckart-Young theorem.

(f) (Implementation, BONUS) Alas! Here we have a bad news. Due to time and resource constraints, we were unable to provide a good and debugged implementation part to this problem. You are free to wander around and play with the starter code and construct simulated datasets (or find real world data sets) where K-SVD recovers the underlying dictionary and/or provides a good reconstruction of the data using sparse linear combinations. No submission is required but you are encouraged to share your empirical findings on the piazza post associated with this problem. Feel free to share the datasets/results that you find. Extra credit will be given to those who help put together a nice exploration part of this problem. Given that the original K-SVD paper has over six thousand citations, we are confident that it should be possible to do this, and fun too.

HINT: For those who have taken EE16A in a semester where Massive-IoT-CDMA was used to motivate OMP, can you come up with an example where an eavesdropper listens in without knowledge of the wireless device codes but after lurking for a while and hearing small numbers of devices talk simultaneously can now figure out what all the individual devices' codes were? Alternatively, how about imaging lots of very sparse unknown images but not knowing what the imaging masks were. Can you learn the masks from lots of unknown sparse images being observed using the unknown random masks? Can you replicate the seminal work of Olshausen and Field from 1996 that showed that dictionary learning can help explain mammalian vision? See the Proceedings of the IEEE survey article on dictionary learning 10.1109/JPROC.2010.2040551 for more potential references.

Nope!

Question 3. Dropout in Neural Network and Linear Regression

Dropout is an algorithm to "prevent over-fitting" that is used while training lot of modern neural network implementations. This question will introduce how dropout works and why we can understand dropout as method to do regularization. Consider the simplest neural network – linear regression, which tries to find a vector $\vec{w}$ to minimize

$$l(\vec{w}) = \|\vec{y} - \mathbf{X}\vec{w}\|_2^2 = \sum_{i=1}^{n} \left(y_i - \vec{x}_i^\top \vec{w}\right)^2. \tag{5}$$

Dropout is usually used with stochastic gradient descent. Instead of doing a stochastic gradient (with mini-batch-size 1) step using $\vec{w}_{t+1} = \vec{w}_t - \alpha\nabla_{\vec{w}}(y_i - \vec{x}_i^\top \vec{w})$, we instead apply the formula

$$\vec{w}_{t+1} = \vec{w}_t - \alpha\nabla_{\vec{w}}\left[\left(y_i - \frac{1}{p}(\vec{x}_i \circ \mathfrak{m})^\top \vec{w}\right)^2\right]. \tag{6}$$

Here α is the learning rate, p is a user-specified dropout retaintion rate from the $(0,1)$ interval, $i \sim U(1,n)$ is an index drawn from the uniform distribution, $\circ$ is the element-wise multiplication operator, and $\mathfrak{m}$ is a random vector where $\mathfrak{m}_i \sim \text{Bernoulli}(p)$ iid. In other words, we will randomly mask each feature of a sample to be zero with a probability $1-p$ during SGD steps.

While this problem only deals with dropout on linear regression, SGD steps with dropout seen here can easily generalize to any layers of neural networks that contain trainable parameters. We can randomly mask each element of the output vector from the previous layer to be zero with a probability $1-p$ to regularize the training of the current layer. Dropout is commonly used with fully-connected layers and recurrent neural networks as they often contain a large number of parameters, making them prone to overfitting by nature.

(a) For simplicity, let us say that you are only solving a linear regression problem with only two features, i.e.,

$$\mathbf{X} = [\vec{x}^{(1)} \ \vec{x}^{(2)}].$$

And for concreteness, suppose that we are using $p = \frac{2}{3}$ as the dropout retention rate.

Find a dataset $\mathbf{X}'$ and $\vec{y}'$ so that the normal SGD steps on the dataset $\mathbf{X}'$ and y' is effectively the same as the dropout SGD steps on the original $\mathbf{X}$ and $\vec{y}$. Hint: You can augment the dataset and make it effectively be a weighted linear regression problem by making multiple copies of data points. You can define

$$\mathbf{X}' = \begin{bmatrix} \alpha_1\vec{x}^{(1)} & \alpha_1\vec{x}^{(2)} \\ \alpha_2\vec{x}^{(1)} & \mathbf{0} \\ \mathbf{0} & \alpha_3\vec{x}^{(2)} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{7}$$

and

$$\vec{y}' = \begin{bmatrix} \alpha_1\vec{y} \\ \alpha_2\vec{y} \\ \alpha_3\vec{y} \\ \alpha_4\vec{y} \end{bmatrix} \tag{8}$$

and **find the expression for α_1, α_2, α_3, and α_4, along with how many copies you want to include of each "row" above in your augmented data set. Verify that the SGD steps are indeed the same by showing that the probability of making each kind of update to the weights is the same.**

Let's use the hint and define $\mathbf{X}'$ and $\vec{y}'$ accordingly.

$$\mathbf{X}' = \begin{bmatrix} \alpha_1\vec{x}^{(1)} & \alpha_1\vec{x}^{(2)} \\ \alpha_2\vec{x}^{(1)} & \mathbf{0} \\ \mathbf{0} & \alpha_3\vec{x}^{(2)} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

$$\vec{y}' = \begin{bmatrix} \alpha_1\vec{y} \\ \alpha_2\vec{y} \\ \alpha_3\vec{y} \\ \alpha_4\vec{y} \end{bmatrix}$$

Here are some properties we want:
The probability of keeping both features is $q_1 = p^2 = \frac{4}{9}$.
The probability of keeping one of the features is $q_2 = q_3 = p(1-p) = \frac{2}{9}$.
The probability of keeping no feature is $q_4 = (1-p)^2 = \frac{1}{9}$.
Besides, we want the mean to be $\vec{x}^{(i)}$. Therefore $\alpha_i = \frac{1}{p}$; on the other hand, we want y to stay the same.

$$\mathbf{X}' = \begin{bmatrix} \frac{3}{2}\vec{x}^{(1)} & \frac{3}{2}\vec{x}^{(2)} \\ \frac{3}{2}\vec{x}^{(1)} & \mathbf{0} \\ \mathbf{0} & \frac{3}{2}\vec{x}^{(2)} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \quad \vec{y}' = \begin{bmatrix} \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \end{bmatrix}$$

We also want the corresponding "row" to have the number of copies so that it the proportion of that "row" in the dataset is the same as its desired probability. That is we want

$$\text{"row"}1 : \text{"row"}2 : \text{"row"}3 : \text{"row"}4 = 4 : 2 : 2 : 1$$

Now our dataset becomes:

$$\mathbf{X}' = \begin{bmatrix} \frac{3}{2}\vec{x}^{(1)} & \frac{3}{2}\vec{x}^{(2)} \\ \frac{3}{2}\vec{x}^{(1)} & \frac{3}{2}\vec{x}^{(2)} \\ \frac{3}{2}\vec{x}^{(1)} & \frac{3}{2}\vec{x}^{(2)} \\ \frac{3}{2}\vec{x}^{(1)} & \frac{3}{2}\vec{x}^{(2)} \\ \frac{3}{2}\vec{x}^{(1)} & \mathbf{0} \\ \frac{3}{2}\vec{x}^{(1)} & \mathbf{0} \\ \mathbf{0} & \frac{3}{2}\vec{x}^{(2)} \\ \mathbf{0} & \frac{3}{2}\vec{x}^{(2)} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \quad \vec{y}' = \begin{bmatrix} \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \\ \vec{y} \end{bmatrix}$$

We have already shown that the updates to the weights is what we want. Let's just double check the expectation:

$$\frac{2}{3} \times \frac{3}{2}\vec{x}^{(i)} + \frac{2}{3} \times 0 = \vec{x}^{(i)}$$

This means I have a probability of $\frac{4}{9}$ to keep both scaled data points $\frac{3}{2}\vec{x}^{(i)}$ and a probability of $\frac{2}{9}$ to only keep one of them.

(b) Leveraging what you have learned from the previous part, **explain why** the SGD steps with dropout for linear regression is the same as doing standard SGD without dropout for the following modified loss function

$$l'(\vec{w}) = \mathbf{E}_{\mathfrak{M}}\left[\left\|\vec{y} - \frac{1}{p}(\mathbf{X} \circ \mathfrak{M})\vec{w}\right\|_2^2\right], \tag{9}$$

where $\circ$ is an element-wise matrix multiplication operator and random matrix $\mathfrak{M}_{\mathrm{ij}} \sim \mathrm{Bernoulli}(p)$ iid.

Hint: You can "unroll" the expectation here and show that objective function is the same as traditional linear regression with augmented data like we did in the previous part.

Let's define the augmented data like we did in the previous part. For any case of $\mathfrak{M}$, we can always compute its probability and

$$\begin{aligned}
&l'(\vec{w})\\
=&\mathbf{E}_{\mathfrak{M}}\left[\left\|\vec{y} - \frac{1}{p}(\mathbf{X} \circ \mathfrak{M})\vec{w}\right\|_2^2\right]\\
=&\mathbf{E}_{\mathfrak{M}}\left[\sum_{i=1}^{n}\left\|y_i - \frac{1}{p}(\vec{x}_i \circ \mathfrak{m})^\top \vec{w}\right\|_2^2\right]\\
=&\sum_{i=1}^{n}\mathbf{E}_{\mathfrak{m}_i}\left[\left\|y_i - \frac{1}{p}(\vec{x}_i \circ \mathfrak{m}_i)^\top \vec{w}\right\|_2^2\right]\\
=&p^{n\times d}\left\|\vec{y} - \frac{1}{p}\mathbf{X}\vec{w}\right\|_2^2 + \cdots + (1-p)^{n\times d}\left\|\vec{y} - \frac{1}{p}\mathbf{0}\vec{w}\right\|_2^2
\end{aligned}$$

Now we use common sense and stack those $\vec{y}$s and $\mathbf{X}$s together

$$=\|\vec{y}' - \mathbf{X}'\vec{w}\|$$

where we have $\mathbf{X}' = \begin{bmatrix}\frac{1}{p}\mathbf{X}\\ \vdots \\ \frac{1}{p}\mathbf{0}\end{bmatrix}$ $\quad \vec{y}' = \begin{bmatrix}\vec{y}\\ \vdots \\ \vec{y}\end{bmatrix}$

Particularly, if $\mathbf{X}$ still has two features and one data point, we can simplify this to be:

$$\mathbf{X}' = \begin{bmatrix}\frac{1}{p}\vec{x}^{(1)} & \frac{1}{p}\vec{x}^{(2)}\\ \vdots & \vdots \\ \frac{1}{p}\vec{x}^{(1)} & 0 \\ \vdots & \vdots \\ 0 & \frac{1}{p}\vec{x}^{(1)} \\ \vdots & \vdots \\ \mathbf{0} & \mathbf{0}\end{bmatrix} \quad \vec{y}' = \begin{bmatrix}\vec{y}\\ \vdots \\ \vec{y} \\ \vdots \\ \vec{y}\end{bmatrix}$$

We have shown that the problem can be converted to a OLS problem with augmented data.

(c) To see the fact that dropout is a way of doing regularization, **prove that**

$$\arg\min_{\vec{w}} l'(\vec{w}) = \arg\min_{\vec{w}} \|\vec{y} - \mathbf{X}\vec{w}\|_2^2 + \frac{1-p}{p}\|\mathbf{\Gamma}\vec{w}\|_2^2, \tag{10}$$

where $\mathbf{\Gamma} = \mathbf{diag}(\mathbf{X}^\top\mathbf{X})^{1/2}$ — i.e. $\mathbf{\Gamma}$ is a diagonal matrix with just the Euclidean norms of the feature vectors along the diagonal. This is saying that applying dropout *on the linear regression problem* is equivalent to doing the ridge regression with a special diagonal Tikhonov regularization matrix. For the effect of dropout on more complex models such as logistic regression or convolution neural networks, it is hard to obtain a closed-form solution. But the idea dropout is equivalent to some kind of regularization still applies.

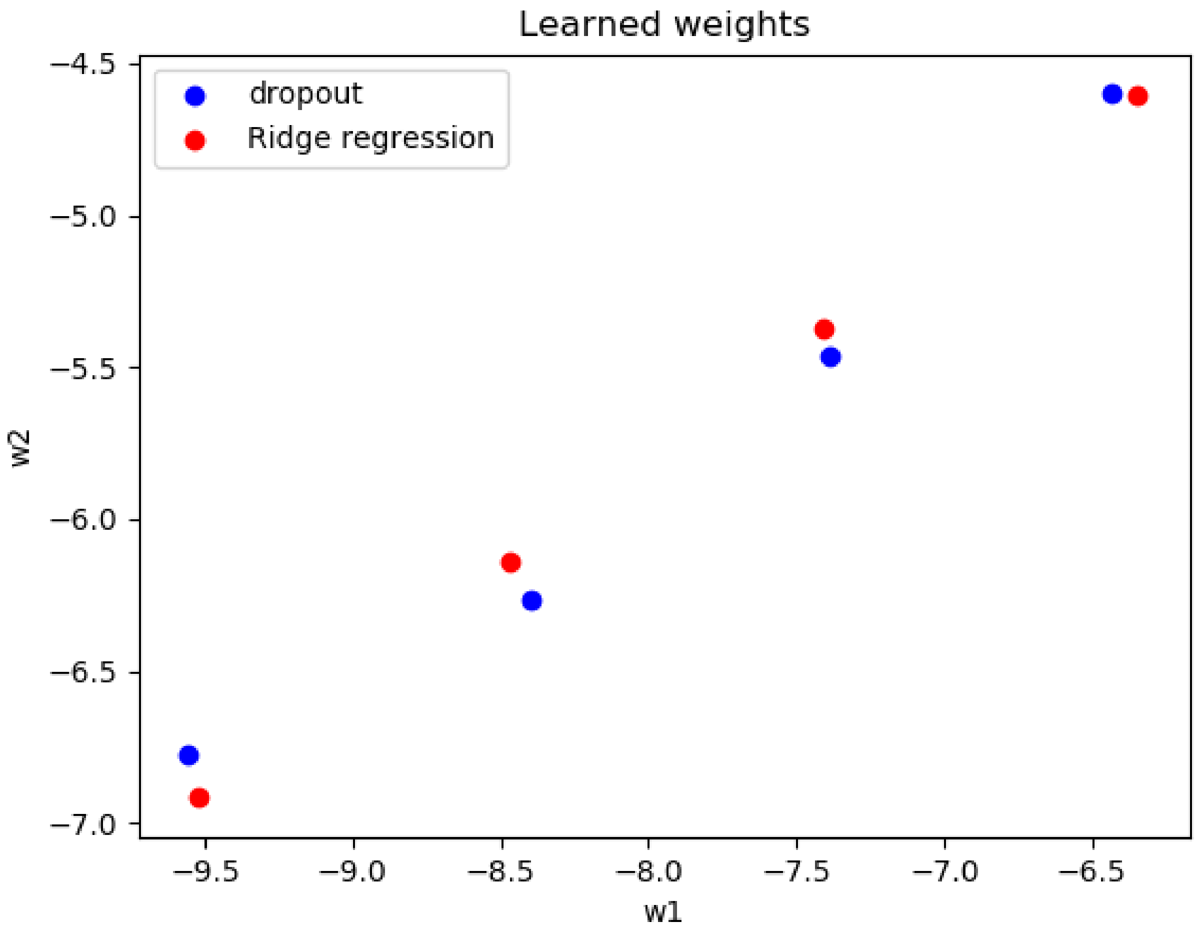
$$\begin{aligned}
&l'(\vec{w})\\
&=\sum_{i=1}^{n}\mathbf{E}_{\mathfrak{m}_i}\left[\left\|y_i - \frac{1}{p}\left(\vec{x}_i \circ \mathfrak{m}_i\right)^\top \vec{w}\right\|_2^2\right]\\
&=\sum_{i=1}^{n}\mathbf{E}_{\mathfrak{m}_i}\left[y_i^2 - 2\frac{1}{p}\left(\vec{x}_i \circ \mathfrak{m}_i\right)^\top \vec{w}y_i + \frac{1}{p^2}\vec{w}^\top\left(\vec{x}_i \circ \mathfrak{m}_i\right)\left(\vec{x}_i \circ \mathfrak{m}_i\right)^\top \vec{w}\right]\\
&=\sum_{i=1}^{n}\left[y_i^2 - 2\vec{x}_i^\top\vec{w}y_i + \frac{1}{p^2}\vec{w}^\top p\vec{x}_i\vec{x}_i^\top\vec{w}\right]\\
&=\sum_{i=1}^{n}\left[y_i^2 - 2\vec{x}_i^\top\vec{w}y_i + \vec{w}^\top\vec{x}_i\vec{x}_i^\top\vec{w} + \frac{1-p}{p}\vec{w}^\top\vec{x}_i\vec{x}_i^\top\vec{w}\right]\\
&=\sum_{i=1}^{n}\left[y_i^2 - 2\vec{x}_i^\top\vec{w}y_i + \vec{w}^\top\vec{x}_i\vec{x}_i^\top\vec{w}\right] + \sum_{i=1}^{n}\left[\frac{1-p}{p}\vec{w}^\top\vec{x}_i\vec{x}_i^\top\vec{w}\right]\\
&=\|\vec{y} - \mathbf{X}\vec{w}\|_2^2 + \frac{1-p}{p}\|\mathbf{\Gamma}\vec{w}\|_2^2
\end{aligned}$$

(d) If you normalize the columns of $\mathbf{X}$ (i.e. scale the features) so that $\|\vec{x}^{(i)}\|_2^2 = 1$ for each column $\vec{x}^{(i)}$, **what is the equivalent Tikhonov regularization matrix if you use dropout? Explain why (or in what context) dropout's induced implicit Tikhonov regularization might be better than the identity matrix that vanilla ridge regression uses when we are working with unnormalized columns of X.**

If I normalize the columns of $\mathbf{X}$, the Tikhonov regularization matrix would be identity matrix.
It's better than the vanilla ridge regression because it can assign different weights to different features.

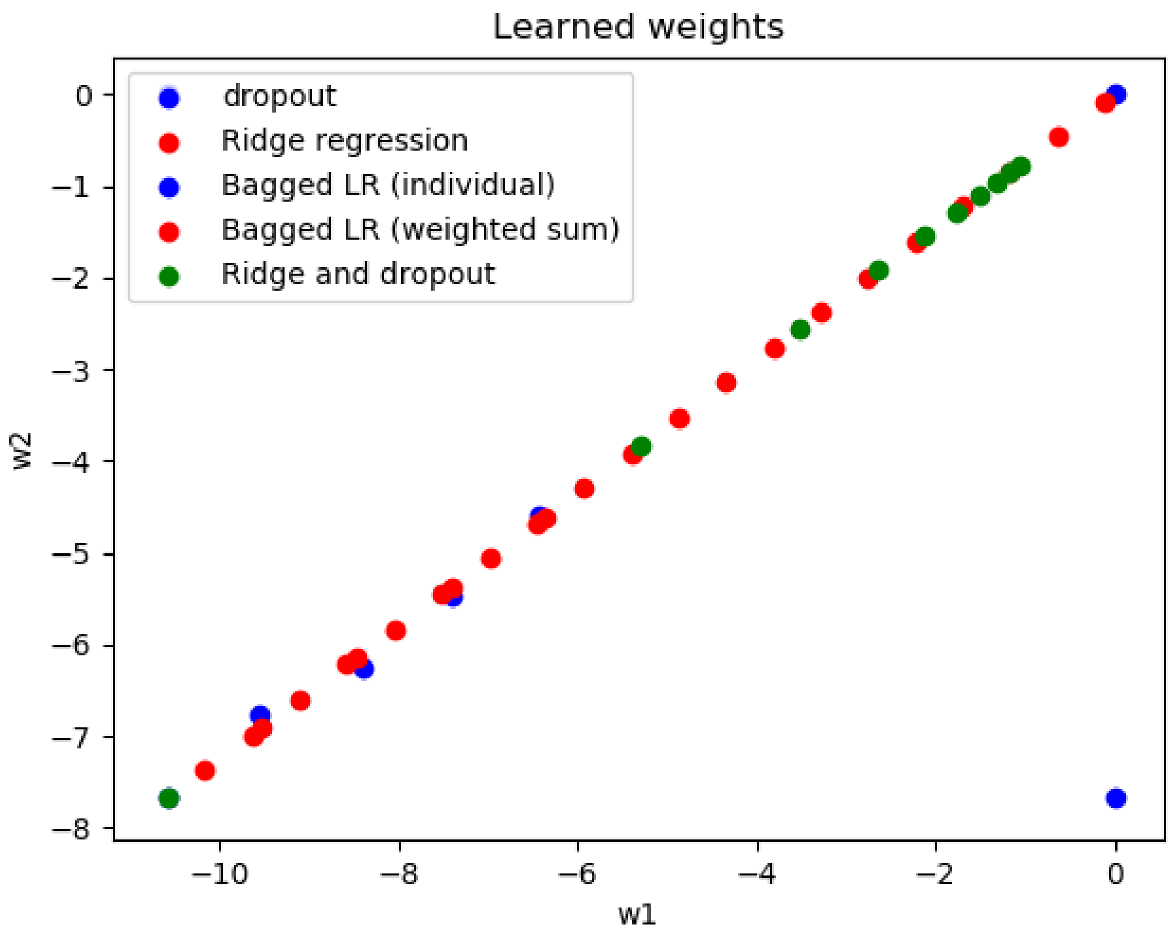
(e) Run the code in Part A. **Explain what the code is doing and the meaning of the plot.**

The code is comparing the weights found by dropout and the weights found by ridge regression. They correlated well as is shown on the graph below. This means dropout of linear regression can be interpreted as ridge regression on the augmented data.



(f) The code in Part B compares dropout with bagged linear regression. Read the code and **explain the differences and similarities between bagged linear regression and linear regression with dropout (implemented using ridge regression in the code).**

Bagged linear regression repetitively resamples data points and run linear regression on the boostrapped data; linear regression with dropout uses the same set out data but setting some weights to zero and ignoring the corresponding data.



Question 4. Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

How do you make a k-nearest neighbor classifier from a 1-nearest neighbor classifier?
You can randomly subsample your data to train approximately k 1-nearest neighbor models and use the ensemble method collectively.