COMP 4360 Assignment 2

Nathan Gagné #7855437

April 23rd 2022

1. a)

The main idea of using ridge regression is to reduce sensitivity to noise and overfitting by applying some regularization that reduces the effect of weights in $\mathbf{w}$. We introduce some amount of bias to reduce the variance of our model. This may result in the model performing slightly worse during training, but it will generalize better on testing data.

The related optimization problem is the regularized least-square that can be described as follows:

$$\text{argmin}\frac{1}{2}||\mathbf{X}\mathbf{w}-\mathbf{t}||^2 + \frac{\alpha}{2}||\mathbf{w}||^2, \qquad (1)$$

where $\alpha \geq 0$ is the regularization parameter.

The closed form solution of (1) to solve for weights $\mathbf{w}$ is:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X}+\alpha^2\mathbf{I})^{-1}\mathbf{X}^T\mathbf{t}, \qquad (2)$$

where $\mathbf{I}$ is an identity matrix of an appropriate size.

The use of ridge regression as opposed to a least-squares fit on some data set can be very useful. If the data set is too small, the least-squares line may fit the training data very nicely and minimize the sum of squared error. However, this may result in poor generalization due to overfitting. Ridge regression mitigates this issue by trading training performance for better generalization.

b)

The *kernel trick* consists of mapping data points from an input space into a higher-dimensional feature space. In classifiers, it is used when data that we want to classify is not directly linearly separable. The *kernel trick* implicitly maps the data to a higher dimensional space using some *kernel function* in order for the classes to be linearly separable in the higher-dimension feature space. If a data point belongs to a class in that feature space, it belongs to that same class in the input space.

This idea is summarized nicely by Mercer's theorem:

$$k(\mathbf{x}, \mathbf{z}) = \langle \Phi(\mathbf{x}), \Phi(\mathbf{z}) \rangle_H, \forall \mathbf{x},\mathbf{z} \in X,$$

where H is a higher-dimensional Hilbert space (feature space), and $\Phi : X \to H$ is a mapping function.

If our linear classifier fails to classify datasets that are not linearly separable in the input space, we can use the *kernel trick* to interpret the data in such a way that it become a nonlinear classifier.

Some examples of positive definite kernel functions include:

Linear kernel: $k(\mathbf{x}, \mathbf{z}) = \mathbf{x} \cdot \mathbf{z}$

Polynomial kernel: $k(\mathbf{x},\mathbf{z}) = (\mathbf{x} \cdot \mathbf{z} + v)^p, v \geq 0, p \geq 0$

Gaussian kernel (RBF): $k(\mathbf{x},\mathbf{z}) = \exp(-\gamma \times ||\mathbf{x}-\mathbf{z}||_2^2)$ (Gaussian when $\gamma = \frac{1}{2\sigma^2}$)

Laplacian kernel: $k(\mathbf{x},\mathbf{z}) = \exp(-\frac{d(x-z)}{\sigma})$

*Note: Numerator of the fraction in this equation should be $d(\mathbf{x}-\mathbf{z})$ to match function k, but can't figure it out in markdown.

c)

Given a function: $y = f(x) + \epsilon$, where $\epsilon = N(0, \sigma^2)$ is the error.

We also have a set of training data $S = \{(x_i, y_i)\}$, and we fit a model function $h(\cdot)$.

We can get the predicted value of y_0 by doing

$y_0 = h(x_0) + \epsilon$,

and this equation can be rearranged to be

$\epsilon = y_0 - h(x_0)$ $\epsilon^2 = (y_0 - h(x_0))^2$

According to the Bias-Variance theory, the decomposition of the means squared error ϵ^2 is as follows:

$\mathbb{E}[(y_0 - h(x_0))^2] = \mathbb{E}[\epsilon^2] + (f(x_0) - \bar{h}(x_0))^2 + \mathbb{E}[(h(x_0) - \bar{h}(x_0))^2]$

Noise: $\mathbb{E}[\epsilon^2] = \sigma^2$ describes the amount that the predicted y_0 can differ from the actual $f(x_0)$ due to noise variance.

Bias^2: $(f(x_0) - \bar{h}(x_0))^2 = (\bar{h}(x_0) - f(x_0))^2$ describes the difference between the value our model predicted and the true value, squared.

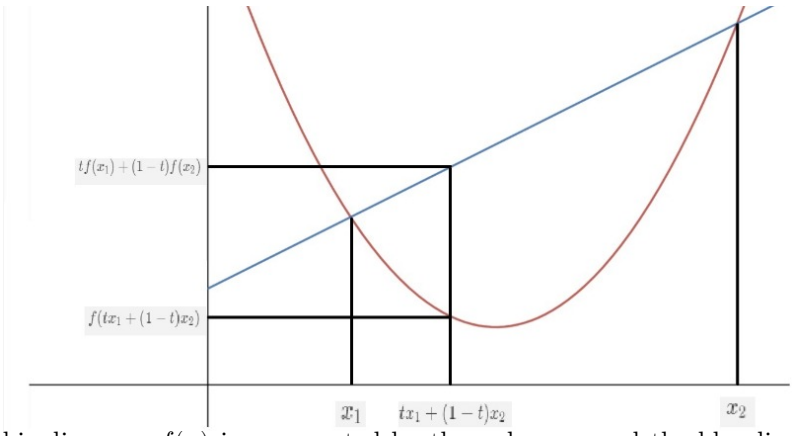
Variance: $\mathbb{E}[(h(x_0) - \bar{h}(x_0))^2]$ describes the variability of $h(x_0)$ when we used different training sets to fit $h(\cdot)$.

Therefore, the Bias-Variance Theory says that we can decompose the error of a theoretical prediction model to:

Error of prediction model = Noise + Bias^2 + Variance

d)

A convex function $f(x)$ is convex on X iff it satisfies *Jensen's inequality.* Here is a diagram to illustrate what we need to know.



In this diagram, $f(x)$ is represented by the red curve and the blue line is the secant line that connects $f(x_1)$ to $f(x_2)$.

$tf(x_1)+(1-t)f(x_2)$: Point on the secant line at time t between $t=0$ (at $f(x_1)$) and $t=1$ (at $f(x_2)$).

$f(tx_1+(1-t)x_2)$): Point on $f(x)$ at that same time t.

Jensen's inequality states that $f(x)$ is convex on X iff for any x_1,$x_2 \in X$,

$tf(x_1)+(1-t)f(x_2) \geq f(tx_1+(1-t)x_2))$ for all $0 \leq t \leq 1$.

In other words, the secant line between $f(x_1)$ and $f(x_2)$ stays above or equal to $f(x)$ in between any two points x_1 and x_2 in X.

It is desirable for an error function to be convex because by definition, a function $f(x)$ that is convex on $(-\infty,+\infty)$ only has one minimum. In that case, the single local minimum would also be the global minimum of $f(x)$. That means that when we want to optimize something by minimizing the error function, we will know that any minimum we converge onto is the global minimum, which is optimal.

2. a)

$\mathbf{x} = [x_0 = 1, x_1, x_2, ..., x_m]^T$ input vector

$\mathbf{w} = [w_0 = b, w_1, w_2, ..., w_m]^T$ weights

$v = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{m} w_i x_i$ pre-activation

$\phi : \mathbb{R} \to \mathbb{R}$ activation function

$y = \phi(v)$ neuron output

$\mathbf{x}$: the input vector contains all the inputs that are being considered by a neuron in a single decision. x_0 is given a value of 1 so that the the total weight of bias b is considered in the dot product for v.

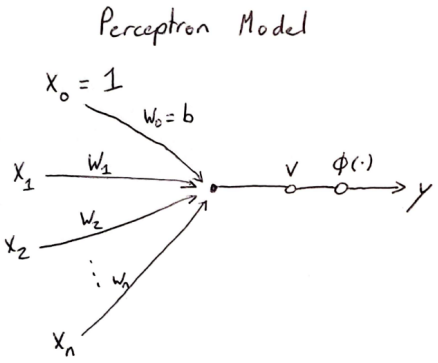
$\mathbf{w}$: the weights vector contains all the weights. Each weight is tied to the input at the same index in $\mathbf{x}$. $w_0 = b$ so that a predetermined bias b is included in the dot product for v.

v: dot product of $\mathbf{w}$ and $\mathbf{x}$. The pre-activation value is used as the input for the activation function

ϕ : the activation function is function mapping $\mathbb{R} \to \mathbb{R}$. Some activation functions output +1 or -1, some have more complex output like this sigmoid function:

$\phi(x) = \frac{1}{1+exp(-ax)}, a < 0$

$y = \phi(v)$: output of the activation function.



b)

We have a three-layer multilayer perceptron (MLP) with one hidden layer of L neurons and one output neuron. This is d-L-1 MLP.

MLPs take an input of dimensionality d and the output is of dimensionality o. It is essentially a mapping: $f : \mathbb{R}^d \to \mathbb{R}^o$.

Perceptron: $y = \phi(v), v = \mathbf{x} \cdot \mathbf{w} = \sum_{i=0}^{d} x_i w_i \quad (1)$

Variables defined and role described in 2. a)

Output Neuron: $y = \phi(v), v = \sum_{i=0}^{L} x_i \alpha_i, \quad (2)$

where α_i is the weight of the connection between the ith hidden layer neuron output and the output neuron.

The output neuron takes the output y of L hidden layer neurons as its inputs x_i and calculates the final output in the same way as the perceptron described above.

Let the output of the ith hidden layer neuron be $y_i^{(H)} \equiv x_i$.

The three-layer neural network can be described as:

$y = \phi(\sum_{i=0}^{L} \alpha_i(y_i^{(H)})) = \phi(\sum_{i=0}^{L} \alpha_i(\phi(\sum_{j=0}^{d} w_{ji} x_j))) \quad (3)$

where w_{ji} is the weight for the connection between the jth input and ith hidden layer neuron.

c)

The Least-Mean-Square (LMS) algorithm performs online gradient descent to find optimal $\hat{w}$ weights by minimizing the objective error function:

$J(\hat{w}(k)) = \frac{1}{2}e^2(k) \quad (1)$

It takes a training set $S_{tr} = \{(x(k), d(k)), k = 1, 2, ..., N\}$ and a learning rate μ as inputs.

The weight vector $\hat{w}$ begins by being set to all zeroes. Then, for each sample in the training set, we compute the instantaneous error, and use that error to perform the update rule which changes $\hat{w}$. Once all samples have been observed, we return the weight vector $\hat{w}$ as output.

The update rule for LMS is as follows:

$$\begin{aligned}\hat{w}(k+1) &= \hat{w}(k) - \mu\ J(\hat{w}(k)) \\ &= \hat{w}(k) + \mu x(k)e(k), \quad (2)\end{aligned}$$

where:

- k is the time-step. Indicates which sample was just observed.
- μ is the learning rate. We assume that this is constant.
- $\hat{w}(k+1)$ is the weight vector at time-step $k+1$. We use a circumflex because the LMS algorithm only produces a per-sample estimate of the weight vector.
- $\hat{w}(k)$ is the weight vector at time-step k.
- $J(\hat{w}(k))$ is the descent direction.
- $x(k)$ is the sample observed at time-step k.
- $e(k)$ is the instantaneous error at time-step k (the error after observing the kth sample)
- We can say that the two lines at (2) are equal because it can be derived that: $J(\hat{w}(k)) = -x(k)e(k)$

d)

The back-propagation learning algorithm for feed-forward neural networks is based on gradient descent.

Some necessary definitions:

Let $S_{tr} = \{(x(n), d(n))\}_{n=1}^{N}$ be a training set.

Let $e_j(n) = d_j(n) - y_j(n) \quad (1)$

be the error produced by the jth output neuron.

Let $\mathcal{E}_j(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n)$ (2)

be the instantaneous error of the network, where C is the set of all indices for the output neurons of the network and n is the index of the sample that was observed.

The goal is to minimize the total error of the network. The algorithm starts by initializing weights.

Forward phase: The forward phase consists of presenting inputs to the network. At the output layer, we calculate the error of the network described in (2).

Backwards phase: The backwards phase starts and we propagate the error back through the network to adapt the weights of each neuron. It achieves this by applying corrections to each of the weights of each neuron in the MLP that is proportional to the partial derivative of the total error with respect to that weight. Derivatives are useful because they tell us the sensitivity of the error function with respect to some variable. Weights that contribute more to the error of the network will have larger partial derivatives, which means they will change more.

Forward and backwards phases are usually repeated several times by shuffling the presentation order of the samples to further train the network.

The weights of the neurons in the hidden layer(s) need to be updated differently as opposed to the neurons in the output layer. This is because the output neuron weights have direct access to the error of the network, so the calculation is rather straightforward. Comparatively, weights in hidden layer neurons do not have direct access to the total error of the MLP. We have to calculate the derivative of the total error with respect to each weight of all hidden layer neurons by considering the cumulative error of all neurons that indirectly connect this weight and the output neuron.

3. a)

Hierarchical clustering is an approach to clustering that starts with N singleton clusters and outputs a partition containing a single cluster. Comparatively, divisive clustering algorithms begin with a partition with a single cluster containing all data points and outputs a partition with the number of clusters that were found.

Hierarchical clustering may be more appealing when an iterative visualization of the cluster merging process is desired. Dendograms are much easier to produce using a bottom-to-top approach. The option to examine the cluster merging process iteratively makes it flexible and allows for interaction with the user. Also, the approach of starting from N clusters and slowly merging them ensures that they will form sound clusters as opposed to some divisive algorithms that can be more volatile when there is randomness involved and make decisions that are suboptimal in the short term.

b)

To perform the hierarchical agglomerative clustering discussed in class notes, we start with an N×N matrix called **D**, where N is the number of points in our data set. Every individual point in the data set begins as its own cluster C_i. Each entry D_{ij} of that matrix is the distance between point C_i and C_j, $d(C_i, C_j)$, with zeroes on the diagonal. Only one side of the diagonal is used, since the same information is stored on both sides.

The equation to calculate distance between clusters is the following:

$d(C_i, C_j) = min_{x_l \in C_i, x_k \in C_j} d(x_l, x_k) \quad (1)$

*I think calling it d_{min} in the notes is a bit redundant

For each iteration:

First find the minimum entry in the matrix. In other words, $min(d(C_i, C_j))$. When that is found, those two clusters are merged into a larger cluster. The two rows/columns are removed from **D** and replaced by a row/column representing the merged cluster. Then, all distances in other rows/columns are updated to reflect the merged cluster by using (1) and comparing each other cluster to the new merged cluster.

Agglomerative clustering ends when there is only one cluster remaining comprised of the entire data set.

c)

Symbol definitions: - Let $X = \{x_1, x_2, ..., x_N\} \subset \mathbb{R}^d$ be an input dataset

- A partition $P = \{C_1, ..., C_K\}$ of X is a grouping in K subsets
- Each $C_i \in P$ is called a cluster, and:
- $C_i \neq \emptyset, i = 1, ..., K$, "every cluster is non-empty"
- $\cup_{i=1}^{K} C_i = P$, "the union of all clusters is the whole partition"
- $C_i \cap C_j = \emptyset, i \neq j, i, j = 1, ..., K$, "the intersection of any two distinct clusters is the empty set"
- A partition P has K cluster representatives $\mu_j, j = 1, ..., K$

Given all of this, the objective function minimized by the k-means algorithm is the following:

$$J(P) = \sum_{j=1}^{K} \sum_{x_i \in C_j} ||x_i - \mu_j||_2^2$$

d)

Input: Data $X = \{x_1, ..., xn\}$, the number of clusters k, MAX number of allowed iterations

Output: A partition $P = \{C_1, ..., C_K\}$

1: $t = 0$, $P = \emptyset$

2: Randomly initialize cluster representatives μ_i, $i = 1, ..., K$

3: **loop**

4: $t+ = 1$

5: **Assignment Step:** assign each sample x_j to the cluster with the nearest centroid.

6: $C_i^{(t)} = \{x_j : d(x_j, \mu_i) \leq d(x_j, \mu_h) \text{ for all } h = 1, ..., K\}$

7: **Update Step** update the representatives

8: $\mu_i^{(t+1)} = \frac{1}{|C_i^{(t)}|} \sum_{x_j \in C_i} x_j$

(The cluster representative μ_i is updated to be the new centroid of C_i)

9: Update the partition with the modified clusters: $P^t = \{C_1^{(t)}, ..., C_K^{(t)}\}$

10: **if** $t \geq$ MAX OR $P^t = P^{t-1}$ **then**

11: **return** P^t

12: **end if**

(The end condition of the algorithm is if the max number of iterations is reached or if the resulting partitions in two subsequent iterations are identical)

13: **end loop**

e)

I am assuming that the k-means algorithm implementations that the two users run are identical

This phenomenon is not unexpected because depending on the implementation, the k initial cluster representatives are decided at random. This causes the initial clusters after the first assignment step to be different, may lead to different cluster representatives in future iterations, and final partitions can be different. The lower the maximum number of iterations MAX is, the more likely this is to occur because the algorithm does not have the number of iterations required to converge on identical partitions for the two trials. However, even if both trials terminate because the partitions for two subsequent iterations are identical, the two output partitions are not guaranteed to be identical.

In order to mitigate this occurrence, the implementation could select the k initial cluster representatives in some sort of deterministic way. For example, you could evenly space out k indices over all datapoints and select those to be initial cluster representative. This would give two users the same output partitions while lowering the risk of selecting cluster representatives that are all in the same region (for example, if a data set has nearby datapoints next to each other in the input list).