CS 1810 Spring 2025 Midterm Review Session

1 Regression

1.1 Linear Regression and Loss Minimization

In supervised learning, we have some data $\{(\boldsymbol{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$ and seek to model/predict the output $y^{(n)}$ based on the inputs $\boldsymbol{x}^{(n)}$. We call the output our **target/response** variable and the inputs our **features/predictors**. In **linear regression** we form a prediction $\hat{y}$ based on a linear combination of the features as such:

$$\hat{y} = \sum_{d=1}^{D} w_d x_d = \boldsymbol{w}^\top \boldsymbol{x}$$

The weights $\boldsymbol{w}$ are our model **parameters**, and our task in linear regression is to learn these parameters. To solve for the optimal parameters, we need to define a **loss function** that depends on $\boldsymbol{w}$. The loss function captures how well our model fits the true data through outputting what is essentially an error metric. For ordinary least squares (OLS) regression, we use the **least squares** loss function:

$$\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{2} \sum_{n=1}^{N} \left(y^{(n)} - \boldsymbol{w}^\top \boldsymbol{x}^{(n)} \right)^2 = \frac{1}{2} ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}||^2$$

We want to find the value of $\boldsymbol{w}$ that minimizes the loss function. To compute this, we expand the loss function, compute its derivative, and solve for $\boldsymbol{w}$ which sets the derivative equal to 0:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) &= \frac{1}{2} \left(\boldsymbol{y}^\top \boldsymbol{y} - 2\boldsymbol{y}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} \right) \\
\implies \nabla_{\boldsymbol{w}} \mathcal{L} &= -\boldsymbol{X}^\top \boldsymbol{y} + \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} \\
\implies \boldsymbol{w}^*_{OLS} &= (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}
\end{aligned}$$

Note that we need $\boldsymbol{X}^\top \boldsymbol{X}$ to be invertible.
This procedure can be generalized to arbitrary parametric models and loss functions. The name of the game here is to fit the model parameters through solving for the values that minimize the loss function. Keep in mind that an analytical solution doesn't always exist.

1.2 Probabilistic Regression and MLE

In the previous subsection, we didn't make any assumptions on the joint distribution that our data were drawn from. We now suppose that our data is generated as such

$$y^{(n)} = \boldsymbol{w}^\top \boldsymbol{x}^{(n)} + \epsilon^{(n)}, \quad \epsilon^{(n)} \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2),$$

The joint likelihood of the data is

$$p(\boldsymbol{y} | \boldsymbol{X}, \boldsymbol{w}, \sigma^2) = \prod_{n=1}^{N} \frac{\exp\left(-\frac{(y^{(n)} - \boldsymbol{w}^\top \boldsymbol{x}^{(n)})^2}{2\sigma^2} \right)}{\sigma\sqrt{2\pi}}$$

Hence, the log likelihood is

$$\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \sigma^2) = -\frac{1}{2\sigma^2}\sum_{n=1}^{N}(y^{(n)} - \boldsymbol{w}^\top \boldsymbol{x}^{(n)})^2 - N\left(\log(\sigma) + \frac{1}{2}\log(2\pi)\right)$$

For simplicity, let's assume σ^2 is known so that we only care about estimating $\boldsymbol{w}$. To learn these parameters, we seek to maximize the log-likelihood. This is called **maximum likelihood estimation**. Observing the above equation, we see that clearly the solution to this problem is the same as the OLS estimator!

1.3 Basis Regression

By default, linear regression only allows us to model linear relationships between the raw inputs $\boldsymbol{x}$ and the output y. However, we can apply **basis transformations** to our inputs in order to model non-linear relationships. Say the true data generating relationship is $f(x) = 1 + x^2$: using the basis transformation $\phi(x) = [1\ x\ x^2]^\top$ would let us perfectly model the function with $\boldsymbol{w} = [1\ 0\ 1]^\top$ while a linear regression would model the relationship very poorly. In general, we use a basis function $\phi : \mathbb{R}^D \to \mathbb{R}^M$ to formulate a prediction as such:

$$\hat{y} = \sum_{m=1}^{M} w_m \phi_m(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x})$$

Observe that we just treat the transformed inputs in the same way that we treated the raw inputs in the simple linear regression from earlier. To merge the bias term, we can define $\phi_1(\boldsymbol{x}) = 1$. Some examples of basis functions include polynomial $\phi_m(x) = x^m$, Fourier $\phi_m(x) = \cos(m\pi x)$, and Gaussian $\phi_m(x) = \exp\{-\frac{(x-\mu_m)^2}{2s^2}\}$.

1.4 Nonparametric Regression

A **nonparametric** regression method makes no assumptions about the structure underlying the data. ***k*-Nearest Neighbors** (k-NN) is one of these methods since for a fixed value of k that you choose, there are no other parameters that the model learns. Here is a rundown of the k-NN Algorithm:

1. Let $\boldsymbol{x}^*$ be the point that we would like to make a prediction about. Let's find the k nearest points $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(k)}\}$ to $\boldsymbol{x}^*$, based on some predetermined distance function.
2. Denote the true y values of these k points as $\{y^{(1)}, \ldots, y^{(k)}\}$.
3. Output our prediction $\hat{y}^*$ for our point of interest $\boldsymbol{x}^*$:

$$\hat{y}^* = \frac{1}{k}\sum_{n=1}^{k} y^{(n)}$$

Kernel regression is considered to be a smoother, more general extension of k-NN. In Kernel regression, we want to take a *weighted average* of all the points in our training data when forming a prediction for an unknown point. Intuitively, we want to weigh points that are "closer" to our

unknown point of interest *more heavily* than points that are "farther" away. Define $K(\boldsymbol{x}^*, \boldsymbol{x})$ as our kernel function, which we will use to weigh each point. The kernel function $K(\boldsymbol{x}^*, \boldsymbol{x})$ should be *larger* for a point $\boldsymbol{x}$ closer to our point of interest $\boldsymbol{x}^*$ than a point $\boldsymbol{x}$ farther away. Importantly, the value of $\boldsymbol{x}$ that results in the largest value of $K(\boldsymbol{x}^*, \boldsymbol{x})$ should be $\boldsymbol{x}^*$ itself:

$$\arg\max_{\boldsymbol{x}} K(\boldsymbol{x}^*, \boldsymbol{x}) = \boldsymbol{x}^*$$

Here is a rundown of kernel regression:

1. Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}\}$ (and their corresponding y values) be *all* of the N points comprising our training data set.
2. Let $\boldsymbol{x^*}$ be our point of interest that we want to make a prediction for. We make our prediction as follows:
$$\hat{y}^* = \frac{\sum_{n=1}^{N} K(\boldsymbol{x}^*, \boldsymbol{x}^{(n)}) y^{(n)}}{\sum_{n=1}^{N} K(\boldsymbol{x}^*, \boldsymbol{x}^{(n)})}$$

The denominator term normalizes the sum of our weights to equal 1. Compared to the k-NN algorithm, the kernel regression uses all the points in the dataset to predict rather than just the k nearest.

1.5 Exercises

1. Consider the following data $\mathcal{D} = \{(x, y)\} = \{(1, 2), (2, 4), (3, 6), (4, 8)\}$. You are asked to apply OLS linear regression to find the optimal weights $\boldsymbol{w}$, with the following basis transformation applied to the data:

   $$\phi(x) = [x, x]^T$$

   Can you find a unique solution for $\boldsymbol{w}$? Why or why not?

   Solution: It is not possible to find a unique analytic solution for $\boldsymbol{w}$. With the basis transformation applied, the matrix $\boldsymbol{X}$ resulting from applying $\phi(x)$ to our data is singular, and as a result, $\boldsymbol{X}^T\boldsymbol{X}$ is not invertible, which is a requirement for our OLS analytic solution.

   This can happen when a column in our data is a perfect linear combination of the other columns, which our basis transformation above brings about. Note that this can also happen when a dataset has more features than datapoints, for the same reason.

2. For k-NN regression with N datapoints, how do bias and variance change as you increase k from 1 to N?

   Solution: At very small values of k, the model perfectly fits to every data point and thus has high variance and low bias. At very large values of k, for instance $k = N$, the model predicts the same value (the mean of the whole dataset), and so it has low variance and high bias. As it increase between, the variance decreases and the bias increases.

2 Classification

2.1 Perceptron and Gradient Descent

We now move on to **classification**.

1. Goal: Given an input vector $\boldsymbol{x}$, assign it to one of K discrete **classes** C_k. Examples of classes include star types or spam vs. regular emails.

2. Strategy: Divide our input space into *disjoint* (i.e., no overlap) **decision regions** whose boundaries are called **decision boundaries/surfaces**. Each decision region corresponds to being assigned to a certain class: there should be K decision regions if we are working with K discrete classes.

In **binary linear classification**, we are working with two classes divided by a linear separator in our feature space. We will denote the two classes as -1 and 1 (note that in other situations, we might use 0 and 1). **Perceptron** is a non-probabilistic binary linear classification algorithm that uses a **discriminant function** h to assign a given observation to a specific class, based on $\boldsymbol{x}$. Perceptron uses the sign of the discriminant to predict as such:

$$\hat{y} = \text{sign}(h(\boldsymbol{x}; \boldsymbol{w}, w_0)) = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x} + w_0),$$

where $\text{sign}(z) = 1$ if $z \geq 0$, and $\text{sign}(z) = -1$ if $z < 0$. To fit the optimal weights, we must first define a loss function. By construction of $\hat{y}$, we note that $h(\boldsymbol{x}^{(n)}; \boldsymbol{w}, w_0)y^{(n)}$ is positive when $\hat{y}^{(n)} = y^{(n)}$ and negative when $\hat{y}^{(n)} \neq y^{(n)}$. Hence, we use the hinge loss / ReLU:

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}) &= \sum_{n=1}^{N} \text{ReLU}(-h(\boldsymbol{x}^{(n)}; \boldsymbol{w}, w_0)y^{(n)}) \\ &= -\sum_{n:y^{(n)} \neq \hat{y}^{(n)}} (\boldsymbol{w}^\top \boldsymbol{x}^{(n)} + w_0)y^{(n)}\end{aligned}$$

We can't analytically solve for the minimizer of this loss, but we can use **gradient descent**. This is a type of numerical method which iteratively updates the weights $\boldsymbol{w}$ at each timestep t based on gradient information from some subset of the data. The general form of gradient descent is

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \mathcal{L},$$

where η is the learning rate (hyperparameter). Different forms of gradient descent use a different subset of the data to compute/approximate the gradient $\nabla_{\boldsymbol{w}} \mathcal{L}$. **Batch gradient descent** uses the entire dataset, **stochastic gradient descent** uses a randomly selected subset, and the perceptron algorithm only uses one observation. This means that perceptron updates its weights one data point at a time as such:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}^{(n)} = \boldsymbol{w}^{(t)} + \eta y^{(n)} \boldsymbol{x}^{(n)},$$

where $\mathcal{L}^{(n)}$ is the loss for data point n.

2.2 Logistic Regression

Like with regression, we can also take a probabilistic view on classification. **Discriminative models** are one of two distinct types of probabilistic classification models. These models focus on modeling $p(y|\boldsymbol{x})$. Here is a rundown of **logistic regression**, a discriminative model for binary classification:

1. Rather than using -1 and 1 for our two classes, we now use 0 and 1.
2. We use the following probabilistic model:

   $$p(y = 1|\boldsymbol{x}, \boldsymbol{w}) = \sigma(\boldsymbol{w}^T \boldsymbol{x}), \quad p(y = 0|\boldsymbol{x}, \boldsymbol{w}) = 1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}),$$

   where σ is the **sigmoid function**:

   $$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

   Note that the sigmoid function maps the real line to $(0, 1)$, corresponding to probabilities.
3. We'll use maximum likelihood estimation to find the optimal weights $\boldsymbol{w}$. The joint likelihood of the data is

   $$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) = \prod_{n=1}^{N} p(y^{(n)}|\boldsymbol{x}^{(n)}, \boldsymbol{w})$$

   Hence, the log likelihood is

   $$\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) = \sum_{n=1}^{N} \log p(y^{(n)}|\boldsymbol{x}^{(n)}, \boldsymbol{w})$$

   We can use the *power trick*, which gives us the following decomposition:

   $$p(y|\boldsymbol{x}, \boldsymbol{w}) = p(y = 1|\boldsymbol{x}, \boldsymbol{w})^y \cdot p(y = 0|\boldsymbol{x}, \boldsymbol{w})^{1-y}$$

   You can also recognize that this is just the PMF of a Bernoulli. Substituting, we have that the log-likelihood is

   $$\begin{aligned}
   \log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) &= \sum_{n=1}^{N} \log \left(p(y^{(n)} = 1|\boldsymbol{x}^{(n)}, \boldsymbol{w})^{y^{(n)}} \cdot p(y^{(n)} = 0|\boldsymbol{x}^{(n)}, \boldsymbol{w})^{1-y^{(n)}} \right) \\
   &= \sum_{n=1}^{N} \left(y^{(n)} \cdot \log p(y^{(n)} = 1|\boldsymbol{x}^{(n)}, \boldsymbol{w}) + (1 - y^{(n)}) \cdot \log p(y^{(n)} = 0|\boldsymbol{x}^{(n)}, \boldsymbol{w}) \right) \\
   &= \sum_{n=1}^{N} \left(y^{(n)} \cdot \log \sigma(\boldsymbol{w}^\top \boldsymbol{x}) + (1 - y^{(n)}) \cdot \log(1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x})) \right)
   \end{aligned}$$

   Maximizing this log-likelihood is equivalent to minimizing the negative log-likelihood. Hence, we can treat $-\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})$ as our loss function, take the gradient with respect to $\boldsymbol{w}$ and find the weights that set the gradient equal to zero. There is no analytical solution, so we use gradient descent to solve for the optimal $\boldsymbol{w}$.

Recall that this formulation can be extended to multi-class classification through using the softmax function instead of the sigmoid function.

2.3 Generative Models and Naive Bayes

Generative Models are the other type of probabilistic classification model. These incorporate the complete data generation process into its model via the *joint distribution* $p(\boldsymbol{x}, y)$ of the class y and the input data point $\mathbf{x}$. Note that Bayes' Rule allows us to decompose the joint as such:

$$p(\boldsymbol{x}, y) = p(\boldsymbol{x}|y)p(y)$$

Hence, we can impose distributions for $p(y)$, the **class prior**, and $p(\boldsymbol{x}|y)$, the **class-conditional distribution** to specify a complete generative model. To form predictions, we pick the class C_k that maximizes $p(y = C_k|\boldsymbol{x})$. Note that the definition of conditional probability tells us that

$$p(y|\boldsymbol{x}) \propto p(\boldsymbol{x}, y) = p(\boldsymbol{x}|y)p(y),$$

so we have all the information we need to predict using our generative model.
Naive Bayes is one type of generative model for classification. The key property of this general model is the "naive" assumption that each feature x_d is *conditionally independent* given the class of the target. This can be mathematically formulated as such:

$$p(\boldsymbol{x}|y = C_k) = \prod_{d=1}^{D} p(x_d|y = C_k)$$

We use Naive Bayes to limit the number of parameters needed to specify our model. If our features were dependent on each other, then we would need to explicitly model this dependence using additional parameters. Here we only need the class prior $p(y)$ and the conditional probabilities $p(x_d|y = C_k)$ to specify our generative model.

2.4 Exercises

1. In the softmax setting, why must $p(y = k \mid \mathbf{x})$ sum to 1 across all classes k? Explain in words how softmax generalizes logistic regression from the binary case to the multi-class case. Suggest a method to make decision boundaries nonlinear without changing the form of the softmax function.

   Solution: The sum across all classes is necessarily 1 because the output of the softmax function is a probability distribution over all classes. Softmax is the multi-class analog of the sigmoid used in logistic regression since we have multiple exponentiated linear functions normalized by their sum. To make decision boundaries nonlinear, we can use basis functions to transform the input space before applying softmax.

2. You are comparing two classifiers for a multi-class classification problem: Naive Bayes and logistic regression. Both achieve roughly the same training loss, but on the validation set Naive Bayes does much worse. Considering the assumptions behind Naive Bayes, why might this be the case?

   Solution: Naive Bayes can underperform when features are strongly correlated since it models them as conditionally independent given the class. Meanwhile, logistic regression

does not make any assumptions like this.

3 Model Selection

3.1 Bias-Variance Decomposition

For a given supervised learning task, we ultimately want to select a model that predicts well on unseen data. One thing we can do is examine the mean squared errors of several candidate models on a validation set. We can also reason about this more generally through analyzing the expected squared error $\mathbb{E}_{D,y|\boldsymbol{x}}[(y - f_D(\boldsymbol{x}))^2]$ of our trained model f_D. In particular, the **bias-variance decomposition** tells us that

$$\mathbb{E}_{D,y|\boldsymbol{x}}[(y - f_D(\boldsymbol{x}))^2] = \mathbb{E}_{y|\boldsymbol{x}}[(y - \mathbb{E}_{y|\boldsymbol{x}}[y])^2] + \left(\mathbb{E}_{y|\boldsymbol{x}}[y] - \mathbb{E}_D[f(\boldsymbol{x})]\right)^2 + \mathbb{E}_D[(\mathbb{E}_D[f(\boldsymbol{x})] - f_D(\boldsymbol{x}))^2]$$

Using more condensed notation, this can also be expressed as

$$\mathbb{E}_{D,y|\boldsymbol{x}}[(y - f_D(\boldsymbol{x}))^2] = \underbrace{\mathbb{E}_{y|\boldsymbol{x}}[(y - \overline{y})^2]}_{\text{Noise}} + \underbrace{\left(\overline{y} - \overline{f}(\boldsymbol{x})\right)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_D[(\overline{f}(\boldsymbol{x}) - f_D(\boldsymbol{x}))^2]}_{\text{Variance}}$$

Let's review how to interpret each of these three terms:

1. **Noise**: This captures the inherent variability in the values that you'd get from repeatedly sampling $y \sim p(y|\boldsymbol{x})$.
2. **Bias**: This captures how far off your average prediction (over all possible training datasets) of y from $\boldsymbol{x}$ is from the average value of y given $\boldsymbol{x}$.
3. **Variance**: This captures how much your prediction of y from $\boldsymbol{x}$ varies depending on the dataset D that your model f_D was trained on.

We usually describe the tradeoff between bias and variance through the concepts of **overfitting** and **underfitting**. An overfit model tends to have low bias but high variance. An example would be a degree 1000 polynomial regressor. A common symptom of an overfit model is a low train MSE but a high test MSE. On the other hand, an underfit model tends to have a high bias but low variance. An extreme example is using a constant model such as $f_D = 0$. As far as the train and test MSEs, we'd likely see these both be similar but high.

3.2 Improving Models: Ensemble Methods and Regularization

Given the bias-variance decomposition, we can naturally think of improving our model through either reducing bias or reducing variance. **Ensembling** is a general approach that takes advantage of multiple models to obtain better predictive accuracy than with a single model alone.

In **bagging**, we resample "new" datasets from the training set (this is called *bootstrapping*), fit a strong learner (relatively flexible/complex model) on these datasets, and combine the predictions from the strong learners. This method works by reducing the variance of these strong learners, which already have a relatively low bias. An example is the random forest, which is an average of predictions from decision trees!

In **boosting**, we train a series of weak learners (relatively simple models) sequentially. After training the i^{th} model on a sample of the training set, we train the $(i+1)^{th}$ model on a new sample based on the performance of the i^{th} model. The idea is that the examples classified incorrectly in the previous step receive higher weights in the new sample, encouraging the new model to focus on those examples. We then combine the predictions of these models through a weighted average. This method works by reducing the bias of the weak learners, which already have low variance. The Adaboost algorithm is an example of boosting.

Another commonly used approach for improving models is **regularization**, which focuses on reducing variance. We will discuss this in the context of linear regression, but it can be generalized to other model classes too. Taking degree d polynomial regression as an example, we note that if we set d too high, we get overfitting. Regularization addresses this problem through adding a **penalty term** $R(\boldsymbol{w})$ to the loss function. Specifically, we have something of the form:

$$\mathcal{L}_{reg}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}||^2 + \lambda R(\boldsymbol{w})$$

We can then solve for the optimal $\boldsymbol{w}$ that minimizes this loss function. With regards to specifying the penalty terms, we've discussed two main ones:

1. **LASSO**: $R(\boldsymbol{w}) = ||\boldsymbol{w}||_1 = \sum_{d=1}^{D} |w_d|$. There is no closed-form solution, so numerical methods like cyclic coordinate descent are used. LASSO is unique in that it performs feature selection, i.e. it sets certain w_d equal to 0.
2. **Ridge**: $R(\boldsymbol{w}) = ||\boldsymbol{w}||^2 = \sum_{d=1}^{D} w_d^2$. There is a closed form:

$$\boldsymbol{w}_{ridge} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

3.3 Bayesian Model Selection

When using parametric methods, we sometimes may have some pre-existing belief about the parameters that we capture through a **prior** $p(\boldsymbol{\theta})$. Then in the model selection process, we essentially hope to pick a model that fits the data well and aligns with our prior beliefs. Here is a review on the main components of Bayesian model selection:

- We use Bayes' Rule to combine our prior with the likelihood, forming a **posterior**:

$$p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) p(\boldsymbol{\theta})$$

  If we want a point estimate, we can solve for the value of $\boldsymbol{\theta}$ that maximizes this posterior.

- The **posterior predictive** for new data:

$$p(y^*|x^*, \boldsymbol{X}, \boldsymbol{y}) = \int p(y^*|x^*, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{\theta}$$

  This tells us how to predict the label of a new data point according to the posterior over models obtained by updating the prior with the observed data.

- The **marginal likelihood** of data:

$$p(\boldsymbol{y}|\boldsymbol{X}) = \int p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta}$$

  This tells us how likely the data is, marginalizing over all possible models $\boldsymbol{\theta}$. The marginal likelihood allows us to compare different priors or even different model classes in terms of how well they fit the data—this is precisely model selection!

To summarize, in Bayesian Regression, we encode our assumptions within the prior distribution $p(\boldsymbol{\theta})$, which is updated in the posterior distribution $p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y})$. The posterior functions as our updated beliefs after seeing our data $\boldsymbol{X}, \boldsymbol{y}$, along with the most likely model for that data. The posterior predictive enables us to leverage the data we have seen to create predictions across all potential models (hence, the name). Finally, the marginal likelihood helps describe the likelihood of the data (producing $\boldsymbol{y}$ given $\boldsymbol{X}$ and the model $\boldsymbol{\theta}$) and can be interpreted in model selection as selecting the model that yields the highest likelihood of producing the data distribution (most likely to explain the result).

3.4 Exercises

1. You train a regression model and notice that MSE is nearly zero on training set but high on validation set, with large fluctuations in predictions for slightly different training subsets. What does this suggest about bias and variance? Propose two strategies to address the problem without gathering more data, and discuss how the strategies might help lower validation MSE.

   Solution: Low training loss and high validation loss in the model suggest low bias and high variance, which we call overfitting. Strategies include adding regularization, decreasing model complexity, and using ensemble methods such as bagging. Variance is reduced and we generalize better since we are enforcing smoother or simpler functions, or averaging over multiple models.

2. To mitigate overfitting in your above model, you use regularization. You are debating whether to add an ℓ_1 or ℓ_2 penalty to the weights. Conceptually, how do the learned weights of these two regularization methods differ, and if you suspect that only a few of the features used are actually relevant, which one would you use and how can you choose the regularization strength?

   Solution: The two methods are lasso and ridge. The former can drive some weights to exactly zero while the latter generally shrinks all weights toward zero. If you suspect that only a few features are relevant, you should use lasso. You can choose the regularization strength through cross-validation where a larger λ enforces more shrinkage. Note that variance decreases, but bias may increase.

3. Consider the following settings for choosing hyperparameters for a model: a single 80/20 split between training and validation data, a 5-fold cross-validation, and a leave-one-out (meaning we use N folds) cross-validation. Conceptually, why might the second setting be preferred over the first, and what are some advantages and disadvantages of the third

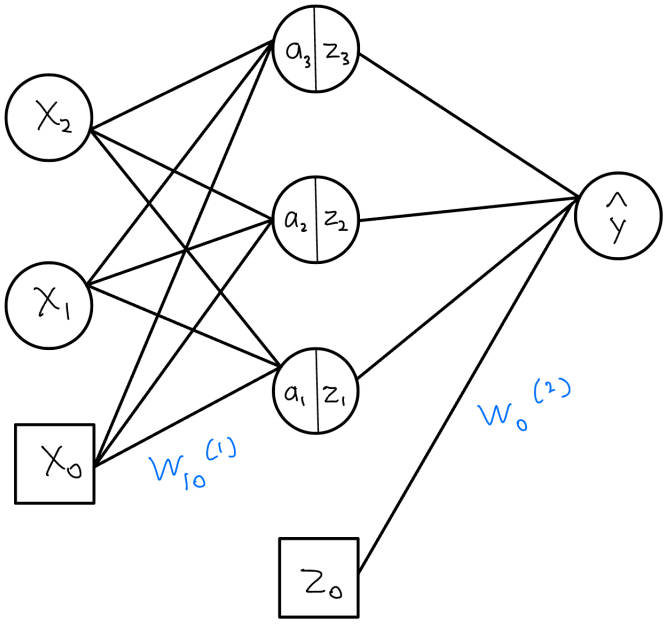
setting?

Solution: The second setting is preferred over the first because it reduces variance by averaging over multiple splits. In addition, leave-one-out cross-validation has the advantage of using almost all datapoints for each training fold which reduces bias since there is more training data, but it can be computationally expensive and have high variance for the test folds.

4 Neural Networks

4.1 Feedforward Networks: Motivation and Setup

Neural networks are a parametric model class that are ubiquitous in modern machine learning. They are widely used because of their ability to learn an adaptive basis, parameterized by weights. Neural networks are very popular because of just how adaptive these bases can be. Namely, we have several **universal function approximation** theorems which essentially state that various neural network architectures can approximate *any* function when taken to be large enough.



Neural networks are essentially a series of connected **layers**, which are sets of **nodes**. The input layer consists of a node for each feature, while the output layer has as many dimensions as the target has. Any intermediate layers in between are called **hidden layers**. Finally, nodes are connected to each other by **connections**, which are parameterized by the model weights. A **feedforward network** is a type of NN in which information only ever flows along the direction of input to output. Mathematically, we can formulate the predictions from a simple NN with one hidden layer as such:

$$\hat{y} = \boldsymbol{w}^{(2)}\boldsymbol{z}, \quad \boldsymbol{z} = h\left(\boldsymbol{a}\right), \quad \boldsymbol{a} = \boldsymbol{W}^{(1)}\boldsymbol{x}$$

We call $\boldsymbol{a}$ the activation and h the **activation function**, which we let be some nonlinear function. Some common activation functions are ReLU, tanh, and sigmoid. This nonlinearity within the neural network is precisely what allows them to be so flexible.

4.2 Neural Network Training

To train neural networks, we use the framework of loss minimization via gradient descent. Thus, the main challenge is to compute the gradient of the loss with respect to all the weights. The overall process of training a neural network is called **backpropagation**. Let's consider the example of training the 3-layer feedforward network from above. Suppose we have the squared error loss function $\mathcal{L}$ and we want to learn the weight $w_{1,0}^{(1)}$. To perform gradient descent, we need to compute $\frac{\partial \mathcal{L}}{\partial w_{1,0}^{(1)}}$. Considering the gradient for data point n, we can use the chain rule as such:

$$\begin{aligned}\frac{\partial \mathcal{L}^{(n)}}{\partial w_{1,0}^{(1)}} &= \frac{\partial \mathcal{L}^{(n)}}{\partial \hat{y}^{(n)}} \cdot \frac{\partial \hat{y}^{(n)}}{\partial z_1} \cdot \frac{\partial z_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_{1,0}^{(1)}} \\ &= (y^{(n)} - \hat{y}^{(n)}) \cdot w_1^{(2)} \cdot h'(a_1) \cdot x_0^{(n)}\end{aligned}$$

Note that we need $x_0^{(n)}$, a_1, and $\hat{y}^{(n)}$ in order to evaluate these gradients. These are all inputs/outputs along the network. The process by which an input $\boldsymbol{x}^{(n)}$ is fed through the network to produce a prediction $\hat{y}^{(n)}$ is called the **forward pass**, and we store all the intermediate values along the way. We leverage this chain rule decomposition of the partial derivatives to efficiently compute them via one **backward pass** through the network. Working with a more general feedforward network, we can decompose each partial derivative as such:

$$\frac{\partial \mathcal{L}^{(n)}}{\partial w_{j,m}^{(l)}} = \frac{\partial \mathcal{L}^{(n)}}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial w_{j,m}^{(l)}} = \left(\sum_{j'=1}^{J_{l+1}} \frac{\partial \mathcal{L}^{(n)}}{\partial a_{j'}^{(l+1)}} \cdot \frac{\partial a_{j'}^{(l+1)}}{\partial a_j^{(l)}} \right) \cdot z_m^{(l-1)},$$

where we assume that layer l of the network has J_l nodes. Note that $\frac{\partial \mathcal{L}^{(n)}}{\partial a_j^{(l)}}$ can be written in terms of the derivatives with respect to the activations from the subsequent layer. This shows that we can compute each *error* $\frac{\partial \mathcal{L}^{(n)}}{\partial a_j^{(l)}}$ starting from the end of the network and going to the start (this is why we call it the backward pass), reusing the precomputed partial derivatives from later in the network to compute the partial derivatives from the layer immediately before. Combined with the input/output values $z_m^{(l-1)}$ that we got from the forward pass, we now have our complete backpropagation method. As seen here, the key idea behind backpropagation is to reuse intermediate results (activations, inputs to each layer, the final output, and the errors) to efficiently compute the gradient of the loss with respect to the network weights.

4.3 Exercises

1. Consider the 3 layer NN pictured above, but now suppose there are $K > 1$ outputs (you can imagine this being a classification problem). At a high level, how does the chain rule for $\frac{\partial \mathcal{L}^{(n)}}{\partial w_{1,0}^{(1)}}$

Solution:

We now have that

$$\frac{\partial \mathcal{L}^{(n)}}{\partial w_{1,0}^{(1)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}^{(n)}}{\partial \hat{y}_k^{(n)}} \cdot \frac{\partial \hat{y}_k^{(n)}}{\partial w_{1,0}^{(1)}}$$

The decomposition of $\frac{\partial \hat{y}_k^{(n)}}{\partial w_{1,0}^{(1)}}$ is analogous to before.

2. Let's examine how the number of parameters in a feedforward NN change with network width and depth. Consider two feedforward neural networks, each with 100 total hidden layer nodes. Both have 10 input nodes and 1 output node. Model 1 has 2 hidden layers, each with 50 nodes, while Model 2 has 10 hidden layers, each with 10 nodes. What are the parameter counts for each? For the sake of simplicity, you can assume there are no bias terms.

   Solution: The connections between the first 2 layers in Model 1 total $10 \times 50 = 500$ weights. The connections between the next 2 layers total $50 \times 50 = 2500$ weights. The connections between the final 2 layers total $50 \times 1 = 50$ weights. So the total number of parameters is 3050 for Model 1.

   For Model 2, we have that the connections between layer l and $l+1$ total $10 \times 10 = 100$ for each of $l = 1, \ldots, 10$. The number of connections between layer 11 and 12 are $10 \times 1 = 10$. Hence, the total number of parameters is 1010 for Model 2, which is noticeably lower than for Model 1.

   One takeaway here is that for a given number of parameters, we can achieve more flexibility via a deep architecture rather than a wide one. If we assume like in this example that all hidden layers in a NN have the same number of nodes, then we see that the number of parameters grows at a rate of $O(\text{depth} \cdot \text{width}^2)$.

5 Support Vector Machines

5.1 Hard Margin Formulation

Support vector machines (SVM's) are classification models that use a linear model of the decision boundary to perform classifications. The idea behind them is that, for all the linear hyperplanes that exist, we want one that will create the largest distance with the training data. At a high level, we define the margin as the minimum distance between a point and our boundary. Larger margins tend to improve generalization error.

Now we'll put this into math. To find a mathematical formula for the margin, we consider a hyperplane of the form

$$\boldsymbol{w}^\top \boldsymbol{x} + w_0 = 0.$$

Furthermore, we can compute the signed distance r between any point $\boldsymbol{x}^*$ and the hyperplane as such:

$$r = \frac{\boldsymbol{w}^\top \boldsymbol{x}^* + w_0}{||\boldsymbol{w}||}$$

Then, note that for a correctly classified data point n, we have $y^{(n)} = +1$ when this distance is positive and $y^{(n)} = -1$ when it is negative. So, we can obtain a positive distance for both kinds of examples by multiplying the above expression by $y^{(n)}$ (which will not change the magnitude since $\|y^{(n)}\| = 1$). Then, we can define the **margin** of the dataset as the minimum such distance over all of our $\boldsymbol{x}^{(n)}$ in our dataset:

$$\min_n \frac{y^{(n)}(\boldsymbol{w}^\top \boldsymbol{x}^{(n)} + w_0)}{||\boldsymbol{w}||}$$

Now to find the optimal $\boldsymbol{w}$, **hard margin** SVM's solve the following problem:

$$\arg\max_{\mathbf{w}, w_0} \frac{1}{||\mathbf{w}||} \min_n y^{(n)}(\mathbf{w}^\top \mathbf{x}^{(n)} + w_0)$$

Now assume the data are completely linearly separable. This implies that the margin of the dataset is positive. Observing that $\boldsymbol{w}$ and w_0 are invariant to changes of scale, it is without loss of generality to impose $\min_n y^{(n)}(\mathbf{w}^\top \mathbf{x}^{(n)} + w_0) \geq 1$. This lets us write the optimization problem as:

$$\arg\min_{\mathbf{w}, w_0} \frac{1}{2}||\mathbf{w}||^2 \quad \text{s.t. } \forall n \; y^{(n)}(\mathbf{w}^\top \mathbf{x}^{(n)} + w_0) \geq 1$$

5.2 Soft Margin Formulation

Recall that towards the end of the hard margin derivation, we imposed the assumption that the dataset is linearly separable. However, this is not always true, and even if the data are linearly separable, it may not be ideal to find a separating hyperplane. In optimizing generalization error, there is a tradeoff between the size of the margin and the number of mistakes on the training data. For the **soft margin** formulation, we introduce a **slack variable** $\xi^{(n)} \geq 0$ for each n to relax the constraints on each example.

$$\xi^{(n)} \begin{cases} = 0 & \text{if correctly classified and not inside margin region} \\ \in (0, 1] & \text{if correctly classified but inside margin region} \\ > 1 & \text{if incorrectly classified} \end{cases}$$

We can now rewrite the training problem for a soft margin formulation to be

$$\arg\min_{\mathbf{w}, w_0, \boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{n=1}^{N} \xi^{(n)}$$

$$\text{s.t. } \forall n \; y^{(n)}(\mathbf{w}^\top \mathbf{x}^{(n)} + w_0) \geq 1 - \xi^{(n)}$$

$$\xi^{(n)} \geq 0$$

We add a regularization parameter C, that controls how much we penalize violating the hard margin constraints. A large C penalizes these violations and thus "respects" the data closely and has small regularization. A small C does not penalize the sum of slack variables as heavily, relaxing the constraint. This is increasing the regularization.

5.3 Dual Form of SVM Training

Let's return to our training problem, specifically from the hard margin formulation. You will not need to know the details of the derivation for the exam, but just be aware that Lagrange multipliers and duality are used to derive the following solution for the optimal $\boldsymbol{w}$:

$$\boldsymbol{w}^* = \sum_{n=1}^{N} (\alpha^*)^{(n)} y^{(n)} \boldsymbol{x}^{(n)}$$

The optimal w_0 can be solved for by using the equation

$$y^{(n)}((\boldsymbol{w}^*)^\top \boldsymbol{x}^{(n)} + w_0) = 1$$

for any point n on the margin boundary. Note that $\boldsymbol{\alpha}^*$ is solved for via quadratic programming methods beyond the scope of the class. Substituting $\boldsymbol{w}^*$ into the discriminant, we see that the SVM classifies a new example $\boldsymbol{x}$ through computing

$$\sum_{n=1}^{N} (\alpha^\star)^{(n)}\, y^{(n)} (\boldsymbol{x}^{(n)})^\top \boldsymbol{x} + w_0^*$$

Based on this, we classify the example as $+1$ if this discriminant value is > 0, and -1 otherwise. We note that the only training points which matter for prediction are those with $(\alpha^*)^{(n)} > 0$. Naturally, we define these points to be the **support vectors**. Under the hard margin formulation, the support vectors lie right along the margin boundaries. Under the soft margin formulation, the support vectors may also lie within the margin region or even be misclassified.

5.4 Kernel Trick

Additionally, the dual has the very nice property that if we use a basis function to map $\boldsymbol{x}$ to a higher dimensional space, this only comes in through the **kernel function**. The training problem is the same as explained earlier, except we replace $(\boldsymbol{x}^{(n)})^\top \boldsymbol{x}^{(n')}$ with

$$K(\boldsymbol{x}^{(n)}, \boldsymbol{x}^{(n')}) = \phi(\boldsymbol{x}^{(n)})^\top \phi(\boldsymbol{x}^{(n')})$$

Similarly, we classify a new example $\boldsymbol{x}$ based on the value of discriminant

$$\sum_{n=1}^{N} \alpha^{\star(n)}\, y^{(n)} K(\boldsymbol{x}^{(n)}, \boldsymbol{x}) + w_0^*.$$

The reason that this is interesting is because we can directly compute the dot product $\phi(\boldsymbol{x}^{(n)})^\top \phi(\boldsymbol{x})$ *without projecting to the higher-dimensional space!* This is known as the **kernel trick**. As long as $K()$ is a valid kernel, the dual training problem can be solved without actually computing ϕ. Note that we can use this trick specifically because the dual gives us an equation for $\boldsymbol{w}^*$ in terms of $\phi(\boldsymbol{x}^{(n)})^T \phi(\boldsymbol{x})$.

5.5 Exercises

1. What happens if we change the constraint in the hard margin formulation to

$$y^{(n)}(\boldsymbol{w}^\top \boldsymbol{x}^{(n)} + w_0) \geq 2$$

for all n?

Solution: The weights may be scaled up, but the decision boundary and predictions remain unchanged. See the SVM section notes for a proof.

2. Suppose you have some solution $\boldsymbol{w}^*$ to the hard margin SVM problem. Find another set of weights $\boldsymbol{w}^{**}$ that defines the same decision boundary, in terms of $\boldsymbol{w}^*$.

Solution: $\boldsymbol{w}^{**} = 2\boldsymbol{w}^*$ or any scalar multiple of $\boldsymbol{w}^*$ works for the same reason as the previous part.