CS 181 Review Session for Midterm 1

Note: We have covered a lot of difficult (but really cool) material so far this semester. It's easy to find yourself in the following situation: you feel that you've learned lots of interesting stuff, and then all of the sudden you feel embarrassed because you have missed some "easy" concept that comes up. This happens all the time, to people that have studied this field seriously. That said, please free to ask any questions that come to mind during the review, and we expect that you will maintain a comfortable environment such that everyone feels good about asking questions.

This review sheet has 3 sections:

- A list of topics particularly relevant to the exam
- An in-depth walk-through of the concepts that we have covered so far this semester
- a glossary of terms

Exam-Relevant Topic List

The best way to prepare for the midterm is to review the lecture slides, two theory psets, section notes, and example midterm questions. Make sure that you understand the most important concepts. These are generally the topics that are covered in detail in lecture.

The midterm will be conceptual and analytical, testing ideas and understanding, and does not involve writing pseudocode. *The material from Lecture 10 (Thursday 2/23) and Lecture 11 Tuesday (2/28) are NOT covered in this midterm.*

You are not expected to memorize complicated formulas such as the PDF for a Gaussian distribution or Beta distribution, complicated matrix cookbook rules, but you should be familiar with methods of probability theory (e.g., Bayes rule) and the various models we've studied so far in the course.

Here is a brief list of topics that you could expect to be asked about. This list emphasizes the main focus areas and is not fully inclusive:

- Non-parametric regression vs parametric regression
- Linear regression: least squares loss, solving analytically
- Generative model of linear regression, gradients, maximum likelihood estimation (you can assume we wouldn't ask you to push through a major derivation without giving you some help)

- Basis functions (general idea, not specific versions)

- Use of validation for model selection and to avoid over-fitting

- Bias-variance tradeoff (not full derivation, but understanding)

- The role and mathematical form of major types of regularization. Particularly when used with linear regression problems

- Idea of ensemble methods

- Bayesian methods: terminology (e.g., "posterior"), MAP, posterior predictive, use of conjugate distributions (Beta/Bernoulli, Normal-Normal), interpretation of estimators, Bayesian linear regression (you can assume we wouldn't ask you to push through a major derivation without giving you some help). Idea of posterior predictive Bayesian LR. [Bayesian model selection is out of scope.]

- Binary, linear classification. Least squares vs perceptron (or "hinge") loss. Gradient descent for perceptron loss (and the perceptron algorithm.)

- Decision boundaries, linear and non-linear separators

- Generative classification, via class probability and class conditional distributions (Gaussian and multinomial Naive Bayes). Use of MLE, loss function via negative log likelihood. [Bayesian Multinomial Naive Bayes is out of scope.]

- Metrics: the idea of TP, FP, FN, TN, ROC curve / AUC, Precision, Recall. [we wouldn't expect you to remember the F1-score defn, or the TPR or FPR defns.]

- Generative vs. discriminative, probabilistic models of classification. Logistic regression model, and MLE, gradient descent via chain rule. [We wouldn't expect you to remember the formula for the sigmoid, or derivative of the sigmoid.] Multi-class classification via softmax.

- Role of loss functions (and activations) in defining algorithms for machine learning, stochastic gradient descent (gradients with respect to single examples)

- Neural nets: adaptive non-linear basis functions, basic notation for weights in layers, use of sigmoid activation, use for non-linear separators, role of network architectures (no need to memorize different activation functions)

- Use of neural nets for both classification and regression. Finding gradients on weights using the chain rule. [We wouldn't expect you to remember the formula for the sigmoid, or derivative of the sigmoid.] High level idea of back-propagation. Idea of weight sharing, convolutional nets.

Walkthrough of Concepts

1 Vector Calculus

1.1 Differentiation

You should be familiar with single-variable differentiation, including properties like

$$\begin{aligned}\text{Chain rule: } & \frac{d}{dx}f(g(x)) = f'(g(x))g'(x)\\ \text{Product rule: } & \frac{d}{dx}f(x)g(x) = f'(x)g(x) + f(x)g'(x)\\ \text{Linearity: } & \frac{d}{dx}(af(x) + bg(x)) = af'(x) + bg'(x)\end{aligned}$$

The multivariate case is similar, but now we consider the partial derivative of each pair of input and output dimensions and end up with a matrix:

$$\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\mathbf{x})}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1(\mathbf{x})}{\partial x_n} & \cdots & \frac{\partial f_m(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

(A partial derivative is just like the single-variable derivative, where you treat each variable except the one you're differentiating with respect to as a constant.)

You are not expected to know everything from the matrix cookbook, but you should know the basic formulas such as

$$\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}^\top \mathbf{x}) = \mathbf{x}$$

1.2 Gradient Vector

If f is scalar-valued, its derivative is a column vector we call the gradient vector:

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \cdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

The gradient vector points in the direction of steepest ascent in $f(\mathbf{x})$. This is useful for optimization.

Recall that the local extrema of a single-variable function can be found by setting its derivative to 0. The same is true here, using the condition $\frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{0}$. However, this equation is often intractable. We can search for local minima numerically using **gradient descent**: We start with an initial guess $\mathbf{w}_0$, and then at each step i we update our guess by going in the direction of greatest descent (opposite the direction of the gradient vector)

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \frac{\partial f(\mathbf{w})}{\partial \mathbf{w}}$$

where η is a learning rate. We stop when the value of the gradient is close to 0.

1.3 Bayes' Theorem

This is a central theorem that we will use repeatedly in this course, and is an extension of the product rule.

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

Since we are conditioning on y, y is held constant, and that means $p(y)$ is just a normalization constant. As a result, we often write the above property as

$$p(x|y) \propto p(y|x)p(x)$$

To see this concretely in terms of machine learning: say we observe data D, and we are interested in parameters $\mathbf{w}$. We can write the *posterior distribution* of the parameters given data by by using Bayes' theorem.

$$\underbrace{p(\mathbf{w}|D)}_{\text{posterior}} = \frac{\overbrace{p(D|\mathbf{w})}^{\text{likelihood}}\overbrace{p(\mathbf{w})}^{\text{prior}}}{\underbrace{p(D)}_{\text{evidence}}}$$

Related to this, a maximum a posteriori (MAP) estimate for the parameter $\mathbf{w}$ is the value

$$\arg\max_{\mathbf{w}} p(\mathbf{w}|D) = \arg\max_{\mathbf{w}} p(D|\mathbf{w})p(\mathbf{w})$$

A maximum likelihood estimate (MLE) is the value

$$\arg\max_{\mathbf{w}} p(D|\mathbf{w})$$

Note that the MLE does not require any sort of prior, while the MAP takes the prior into account.

2 High Level Overview of Machine Learning

Broadly speaking, in supervised machine learning, we are focused on using existing data, which generally has inputs and outputs, to make prediction about new inputs. This generally plays out with two types of problems: regression, where we are trying to predict an element in $\mathbb{R}$, and classification, in which we are trying to predict a class label for an input. We will cover other types of machine learning, including unsupervised learning and reinforcement learning later, so stay tuned!

Now, we divide supervised machine learning further. In non-parametric models, we generally keep our training data when we make predictions, since we are not learning parameters from the training data (hence the "non"). A classic example of this is K-nearest neighbors. Generally, the way KNN works is at test time, given an input $\mathbf{x}$, we find the k training data points that are closest to $\mathbf{x}$, and use their output values to make our prediction.

Naturally, this can be problematic if our data is enormous, as it is often is. Thus, this semester, we have focused on **parametric supervised machine learning** models. In these models, there are generally two stages:

1. Learn a set of parameters, which we generally call θ from the training data. Note that θ may be literally one matrix, as is the case in logistic regression, or a set of matrices and other parameters, as is the case in neural networks, or naive bayes. Generally speaking, we will represent the parameters as θ, though we often use $\mathbf{W}$, since these parameters are usually weights.

2. Use the θ from the previous stage to make predictions on new data points. Note that we did not need to store all of the data to make predictions - just the parameters.

3 Linear Regression

The simplest model for regression involves a linear combination of the input variables:

$$h(\mathbf{x}; \mathbf{w}) = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = \sum_{j=1}^{m} w_j x_j = \mathbf{w}^\top \mathbf{x} \tag{1}$$

where $x_j \in \mathbb{R}$ for $j \in \{1, \ldots, m\}$ are the features, $\mathbf{w} \in \mathbb{R}^m$ is the weight parameter, with $w_1 \in \mathbb{R}$ being the bias parameter. (Recall the trick of letting $x_1 = 1$ to merge bias.)

3.1 Linear Basis Function Regression

We allow $h(\mathbf{x}; \mathbf{w})$ to be a non-linear function of the input vector $\mathbf{x}$, while remaining linear in $\mathbf{w} \in \mathbb{R}^d$:

$$h(\mathbf{x}; \mathbf{w}) = \sum_{j=1}^{d} w_j \phi_j(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) \tag{2}$$

where $\phi_j(\mathbf{x}) : \mathbb{R}^m \to \mathbb{R}^d$ denotes the jth term of $\boldsymbol{\phi}(\mathbf{x})$. To merge bias, we define $\phi_1(\mathbf{x}) = 1$.

3.2 Least squares Loss Function

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 \tag{3}$$

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y} = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \tag{4}$$

where $\mathbf{X} \in \mathbb{R}^{n\times m}$, where each row is one data point (i.e. one feature vector) and each column represents values of a given feature across all the data points.

3.3 Regularized Least Squares

To penalize complexity, we add a regularization term to the error function. The total error function becomes:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w} \tag{5}$$

This is known as *Ridge* regression, or L^2 regularization, since we are adding the L^2 norm of the weights to the loss. This is analytically solvable (see the section notes), and our final estimator is given as:

$$\mathbf{w}^* = (\lambda \mathbf{I} + \mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y} \tag{6}$$

Another common form of regularization is *Lasso*, or L^1 regularization (since we take the L^1 norm of the weights). In this case the error function looks like:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i\right)^2 + \lambda \left\|\mathbf{w}\right\| \tag{7}$$

Unfortunately there's no analytic closed form solution to lasso regression, and you can see that the loss under lasso regression is *not* differentiable everywhere, so we need another tactic for optimizing it. Coordinate ascent or subgradients are the usual method, but we don't ask you to know about them.

3.4 Bayesian Linear Regression

In Bayesian Linear Regression, we make a key observation: rather than picking the MLE or MAP estimates for w, we can model the entire distribution over them. Then, we can use the entire distribution over them to make predictions.

Why is this useful? Well, imagine a situation in which the probability density assigned to different values of $\mathbf{w}$ is fairly uniform, with one value $\mathbf{w}^*$ being marginally higher than all of the rest. If we learn just $\mathbf{w}^*$ (though MAP estimation for example), and use just that to make our predictions, we are throwing away information about all of the other possible $\mathbf{w}$ that also carried substantial probability density.

So, rather than picking a particular $\mathbf{w}^*$, we are going to try to model the posterior probability density function of $\mathbf{w}$ itself, and then use the full posterior to make predictions. Let $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, $\mathbf{x}_i \in \mathbb{R}^m$, $y_i \in \mathbb{R}$. Consider the generative model

$$y_i \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x}_i, \beta^{-1}) \tag{8}$$

The likelihood of the data has the form:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \mathcal{N}(\mathbf{y}|\mathbf{X}\mathbf{w}, \beta^{-1}\mathbf{I}) \tag{9}$$

Put conjugate prior on the weights as (assume precision β^{-1} known):

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{m}_0, \mathbf{S}_0) \tag{10}$$

We want a posterior distribution on $\mathbf{w}$ by using Bayes's Theorem, which states that:

$$p(\mathbf{w}|D) \propto p(D|\mathbf{w})p(\mathbf{w}) \tag{11}$$

It turns out (see practice question from section) that our posterior after n examples is also Gaussian:

$$p(\mathbf{w}|D) = \mathcal{N}(\mathbf{w}|\mathbf{m}_n, \mathbf{S}_n) \tag{12}$$

where

$$\mathbf{S}_n = \left(\mathbf{S}_0^{-1} + \beta\mathbf{X}^\top\mathbf{X}\right)^{-1} \tag{13}$$

$$\mathbf{m}_n = \mathbf{S}_n(\mathbf{S}_0^{-1}\mathbf{m}_0 + \beta\mathbf{X}^\top\mathbf{y}) \tag{14}$$

3.5 Posterior Predictive Distributions

So, now we can model the entire posterior distribution over $\mathbf{w}$. This sounds great, but now we have lost a nice property of MAP and MLE: we no longer have a $\mathbf{w}^*$ that we can use to make predictions. Remember, in our summary of parametric supervised machine learning, we stated that the whole point of this process is ultimately to make predictions on new data points. However, we now just have a distribution over $\mathbf{w}$, and cannot simply plug that in to make predictions on new data points. So, since we have a distribution over $\mathbf{w}$ - meaning a probability density for every possible value for $\mathbf{w}$ - we need some way to incorporate all of this information to actually make predictions. To do this, we will essentially use some calculus, and integrate over every possible value of $\mathbf{w}$. This is also called marginalization.

Let $\mathbf{x}$ denote a new data point. Marginalizing over $\mathbf{w}$, we have that

$$p(y|\mathbf{x}, D) = \int_{\mathbf{w}} p(y|\mathbf{x}, \mathbf{w}) p(\mathbf{w}|D) d\mathbf{w} \tag{15}$$

$$= \int_{\mathbf{w}} \mathcal{N}(y|\mathbf{w}^\top \mathbf{x}, \beta^{-1}) \mathcal{N}(\mathbf{w}|\mathbf{m}_n, \mathbf{S}_n) d\mathbf{w} \tag{16}$$

Since each of the terms on the right hand side follows a normal distribution, we can use some math (see lecture 5, slide 33) to find that

$$p(y|\mathbf{x}, D) = \mathcal{N}(y|\mathbf{x}^\top \mathbf{m}_n, \mathbf{x}^\top \mathbf{S}_n \mathbf{x} + \beta^{-1}) \tag{17}$$

This is a nice result. We have shown that when we marginalize over $\mathbf{w}$, we can make predictions on new data points $\mathbf{x}$. Moreover, the output values follow a nice normal distribution with easy-to-calculate parameters.

4 Model Selection

Model selection is methods for choosing between models: how do you know that your model is *correct*?

4.1 Validation

Suppose we have a model for our data $p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \alpha)$ (with parameters $\mathbf{w}$ and some hyperparameter α), such that we can evaluate its performance on some dataset $\mathbf{X}, \mathbf{y}$ using a loss function $\mathcal{L}$. If we can't minimize with respect to α very easily (for example, if it's the regularization coefficient or the number of layers in our neural network), then we have try some values of α to find the best. However, we'd like to compare using different values with a useful metric, so we compare the models using **validation set** loss. 80/20 train/validation is a common split for our data. The more validation data we use, the lower the variance in our estimate of the generalization error, but we might prefer to use more data for training our model.

4.2 Cross Validation

If we want to avoid the tradeoff above, we can use **cross validation** to estimate our generalization error. In **K-fold cross validation** we split the data into K pieces, and sequentially estimate our generalization error by training on $K - 1$ pieces and measuring error on the remaining piece. Then we average over all choices of the left-out piece to get our final estimate. This is primarily used when there's not a lot of data, but training is cheap (since in K-fold cross validation we need to train K times), for example in linear models on small datasets.

4.3 Bias-Variance Decomposition

One of the limitations of statistical modeling is that as we increase the complexity of our models, they tend to **overfit**. Overfitting is when a model has poor generalization ability, so it tends to do well on the training set but has high error outside. Usually overfitting is caused by fitting to noise rather than signals in the data.

The bias-variance decomposition explains this phenomenon. There are two sources of error: **bias**, which is an inability to capture the signal, and **variance**, or error introduced from our choice of training data. Specifically,

$$\text{generalization error} = \underbrace{\text{systematic error}}_{\text{bias}} + \underbrace{\text{sensitivity of prediction}}_{\text{variance}} \tag{18}$$

In mathematical terms, if we have random data D, new input $\mathbf{x} \in \mathcal{X}$ with target $y \in \mathbb{R}$ generated conditional on $\mathbf{x}$, conditional mean $\bar{y} = \mathbb{E}[y|\mathbf{x}]$, a model $h_D : \mathcal{X} \to \mathbb{R}$, and prediction mean $\bar{h}(\mathbf{x}) = \mathbb{E}_D[h_D(\mathbf{x})]$, we can write the generalization error

$$\mathbb{E}_{D,y|\mathbf{x}}[(y - h_D(\mathbf{x}))^2] = \underbrace{\mathbb{E}_{y|\mathbf{x}}[(y - \bar{y})^2]}_{\text{noise}} + \underbrace{(\bar{y} - \bar{h}(\mathbf{x}))^2}_{\text{bias squared}} + \underbrace{\mathbb{E}_D[(\bar{h}(\mathbf{x}) - h_D(\mathbf{x}))^2]}_{\text{variance}} \tag{19}$$

As we increase model complexity, the bias (usually) goes down (because we can model the signal better) and the variance goes up (because we can model randomness in D better). One way we can control what kind of model we use (simple or complex) is to use **regularization**, which penalizes complex models.

4.4 Bagging

Bagging (also confusingly called *bootstrap aggregation*) is an ensemble method used to decrease the variance of estimators. Given a model, we can train multiple copies of the model on subsets of our data (taken by sampling from the original dataset with replacement) and then build a new model that averages together the predictions from the components. Since the entire model has seen all the data, it doesn't increase bias, but it can decrease variance, because unreasonable predictions (from overfitting to a subset) are outweighed by the other models.

Bagging is best for models that already have low bias and high variance, i.e. complex models like decision trees and neural networks. One disadvantage of bagging: your model becomes n times more expensive (both to train and the use!), where n is the number of bagged estimators you use. For expensive models like neural networks this is a serious issue, but for cheap models like decision trees it's not a problem.

4.5 Boosting

Boosting is a technique for turning a weak estimator (high bias) into a strong one by reducing bias. Essentially, we train first weak learner on the dataset, then we reweight the training set by increasing the weight assigned to those examples that were misclassified by the first model. Then we repeat that process a number of times until our performance stops increasing. For prediction, we take the weighted average of weaker learners, where the weights are based on each one's classification accuracy on the training set. This reduces bias by training later models to focus only on "hard" examples.

In practice this tends to actually reduce bias, but not really decrease variance. Also, on some noisy data sets boosting can lead to decreased performance. There are many boosting algorithms, but AdaBoost is the first, and often studied.

5 Classification

In classification, we predict a class label for a new observation from the output space of c discrete classes:

$$\mathcal{Y} = \{C_1, ..., C_c\} \tag{20}$$

The output is represented either as a single integer like $y = 2$, or a one-hot vector with a 1 in the index corresponding to the correct class, and zero elsewhere:

$$[0, 1, 0, 0, 0, ...] \tag{21}$$

A classifier partitions the input space of observation features with decision boundaries/surfaces such as lines, hyperplanes, or spheres. We have explored linear and probabilistic **discriminative** approaches and **generative** approaches for both binary and multiclass classification. In the discriminative approach, we model the conditional $p(y|\mathbf{x}, \mathbf{W})$ (where $\mathbf{W}$ refers to model parameters). In the generative approach we model the joint $p(y, \mathbf{x})$ by way of intermediately modeling $p(y)$ and $p(\mathbf{x}|y)$.

5.1 Linear Discriminative Classification

We directly model $p(y|\mathbf{x}, \mathbf{w})$, where $\mathbf{w}$ are the model parameters. In the case of **Binary Linear Classification**, we are interested in a discriminative function:

$$h(\mathbf{x}; \mathbf{w}, w_0) \tag{22}$$

that directly assigns each observation $\mathbf{x}$ to a positive or negative class. A simple choice is:

$$\mathbf{w}^\top \mathbf{x} + w_0 \tag{23}$$

where $h(\mathbf{x}; \mathbf{w}, w_0) = 0$ is the decision boundary. This classifier predicts $\hat{y} = 1$ if $h(\mathbf{x}; \mathbf{w}, w_0) > 0$ and $\hat{y} = -1$ otherwise. If $\mathbf{x} \in \mathbb{R}^d$, then the decision boundary is a $(d-1)$ dimensional hyperplane.

In linear classification, the data must be **linearly separable** for a classifier to be able to predict without error. Separability can sometimes be helped through the use of basis functions.

5.2 Probabilistic Discriminative Models

5.2.1 Binary Class Logistic Regression

Be careful not to equate ”probabilistic” and ”generative”. We can have probabilistic discriminative models. Consider the binary classification setting. Allow the log probability to be proportional to a linear model h:

$$\ln p(y = 1|\mathbf{x}; \mathbf{w}) \propto \mathbf{w}^\top \mathbf{x} + w_0 = h \tag{24}$$

Threshold the class 1 prediction at $h > 0$ as in the linear model. Now exponentiate:

$$p(y = 1|\mathbf{x}; \mathbf{w}) = \frac{\exp(h)}{\exp(h) + \exp(0)} = (1 + \exp(-h))^{-1} \tag{25}$$

$$p(y = 0|\mathbf{x}; \mathbf{w}) = \frac{\exp(0)}{\exp(h) + \exp(0)} = (1 + \exp(h))^{-1} \tag{26}$$

We have derived the logistic sigmoid activation, which squashes $\mathbb{R}$ to probabilities:

$$\sigma(h) = (1 + \exp(-h))^{-1} \tag{27}$$

We now have a discriminative form for the conditional probabilities:

$$p(y = 1|\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x} + w_0) \tag{28}$$

$$p(y = 0|\mathbf{x}; \mathbf{w}) = 1 - \sigma(\mathbf{w}^\top \mathbf{x} + w_0) \tag{29}$$

We can now pose a loss function:

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{n} \ln p(y_i|\mathbf{x}_i;\mathbf{w}) = -\sum_{i=1}^{n} \ln\left(\sigma(h)^{y_i}(1-\sigma(h))^{1-y_i}\right) \tag{30}$$

$$= \sum_{i=1}^{n} y_i \ln(1+\exp(-h)) + (1-y_i)\ln(1+\exp(h)) \tag{31}$$

where y_i is the true class of input $\mathbf{x}_i$, expressed as 0 or 1. Note that the y_i and $(1-y_i)$ multipliers/exponents are just tricks to cancel out the term we don't care about.

Having written down the loss, we can now take the gradients with respect to the weights and optimize via Gradient Descent. **Really important: Note that although we are taking a probabilistic approach, we are avoiding modeling $p(\mathrm{x}, y)$ but are instead directly computing the conditional $p(y|\mathrm{x};\mathrm{w})$ by optimizing on w.**

Taking the derivative with respect to **w**:

$$\frac{\partial}{\partial \mathbf{w}} \ln(1+\exp(-h)) = -\mathbf{x}_i \frac{\exp(-h)}{1+\exp(-h)} = -\mathbf{x}_i p(y_i=0|\mathbf{x};\mathbf{w}) \tag{32}$$

$$\frac{\partial}{\partial \mathbf{w}} \ln(1+\exp h) = \mathbf{x}_i \frac{\exp(h)}{1+\exp(h)} = \mathbf{x}_i p(y_i=1|\mathbf{x};\mathbf{w}) \tag{33}$$

so that we get

$$\frac{\partial}{\partial \mathbf{w}}\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{n} -y_i\mathbf{x}_i p(y_i=0|\mathbf{x}_i;\mathbf{w}) + (1-y_i)\mathbf{x}_i p(y_i=1|\mathbf{x}_i;\mathbf{w}) \tag{34}$$

5.2.2 Multiclass Logistic Regression

This was covered on your homework. Now we learn a separate weight vector $\mathbf{w}_k$ for the kth class.

$$p(\mathbf{y}=C_k|\mathbf{x};\{\mathbf{w}_\ell\}_{\ell=1}^{c}) = \frac{\exp(\mathbf{x}^\top\mathbf{w}_k)}{\sum_\ell \exp(\mathbf{x}^\top\mathbf{w}_\ell)} \tag{35}$$

We can also write the vector of probabilities as

$$\mathrm{softmax}([\mathbf{x}^\top\mathbf{w}_1,\ldots,\mathbf{x}^\top\mathbf{w}_c]) = \left[\frac{\exp(\mathbf{x}^\top\mathbf{w}_1)}{\sum_\ell \exp(\mathbf{x}^\top\mathbf{w}_\ell)},\ldots,\frac{\exp(\mathbf{x}^\top\mathbf{w}_c)}{\sum_\ell \exp(\mathbf{x}^\top\mathbf{w}_\ell)}\right] \tag{36}$$

Sometimes you may see $\mathbf{w}_k^\top\mathbf{x}$ referred to as activation $\mathbf{z}_k$, so that we are taking the softmax of activations $[\mathbf{z}_1,\ldots,\mathbf{z}_c]$ and $softmax(\mathbf{z})_k$ is the probability that our observation $\mathbf{x}$ is assigned the class k.

5.3 Generative Classification Approach

As stated above, we model $p(\mathbf{x}, y)$ by modeling:

$$p(y) \text{ and } p(\mathbf{x}|y) \tag{37}$$

where class y is generated with $p(y)$ and input $\mathbf{x}$ is generated conditional on the class with probability $p(\mathbf{x}|y)$. In the binary case, the class prior for classes $\{0, 1\}$ can be modeled by by the Bernoulli distribution:

$$p(y; \theta) = \theta^y (1 - \theta)^{1-y} \tag{38}$$

where $\theta = p(y = 1)$. Notice that the exponents are just a convenient notation that cancels out the term for the class that we are not considering. In the multiclass case, we can specify a vector $\boldsymbol{\pi}$ as the prior so that

$$p(y = C_k; \boldsymbol{\pi}) = \pi_k \tag{39}$$

5.3.1 Naive Bayes + Discrete Features

Naive Bayes is one example of a generative classification model that we have covered. It works for both binary and multiclass classification and is used in the case of discrete feature counts. Let's consider **Multinomial Binary Class Naive Bayes** (multiclass is a simple generalization to make once the binary case is made concrete).

Remember that a **multinomial distribution** is a generalization of the Binomial to multiple categories. While we flip a coin in the Binomial setting, we roll a d-sided die in our case many times to observe a sample $\mathbf{x} = (x_1, \ldots, x_d)^\top$ with x_j discrete integers. Here, the vector $\mathbf{x}$ is a histogram over features (dice rolls).

Specifically, assume that given class y, we have probabilities

$$\boldsymbol{\beta}_y = (\beta_{y1}, \beta_{y2}, \ldots, \beta_{yd})$$

of rolling each side of the die.

In Multinomial Binary Class Naive Bayes, we can estimate $\boldsymbol{\beta}_0$ and $\boldsymbol{\beta}_1$ with MLE. This means that we take all observations belonging to class 0, treat each as a vector of counts over features, and sum them together. Then we normalize to get a class 0 distribution over features. We do the same for class 1. (Exercise: write down the math for how you would do this.)

We also estimate priors $\pi_0 = p(y=0)$ and $\pi_1 = p(y=1)$ by counting the proportion of the whole data set that falls under each class. Finally, we consider the likelihood of a new observation $\mathbf{x}$ against our model for class y by considering:

$$p(y|\mathbf{x}) \propto p(y)p(\mathbf{x}|y) \propto \pi_y \left(\prod_{j=1}^{d} (\beta_{yj})^{x_j} \right) \tag{40}$$

where x_j is observation $\mathbf{x}$'s count for the j^{th} feature. We make our prediction by choosing the class that maximizes this quantity. The generalization to more than 2 classes follows the same steps (exercise: write down the math).

Notice that Naive Bayes becomes a linear classifier when expressed in log-space:

$$\ln \pi_y + \sum_{j=1}^{d} x_j \ln(\beta_{yj}) = w_0 + (\mathbf{w}_y)^\top \mathbf{x} \tag{41}$$

where $(\mathbf{w}_y)_j = \ln(\beta_{yj})$ and $w_0 = \ln \pi_y$.

Example: You would like to classify whether a book is written by a Good Author or a Bad Author. You believe that Good Authors tend to use certain collections of words (using each word with a certain frequency relative to the others), and that Bad Authors use their words differently. You have a large collection of books whose author quality (Good / Bad, as determined by Sasha Rush) is already known.

First, we need to estimate how common Good Authors and Bad Authors are in the first place. Let π_{good} be the proportion of books written by Good Authors in your collection. Let π_{bad} be the proportion of bad authors in your collection.

Next, we need the class conditional distribution over features. Here, every word in the shared vocabulary among all authors is our set of features. Treating each book as a histogram of word counts, we add together all histogram vectors for books written by Good Authors, and then we normalize to form a Good Author distribution over words. Call this $\boldsymbol{\beta}_{good}$. We do the same for Bad Authors and find $\boldsymbol{\beta}_{bad}$.

For some new book $\mathbf{x}$ with word counts x_j for the j^{th} word in the vocabulary, we compute the probability that $\mathbf{x}$ was written by a Good Author:

$$p(good|\mathbf{x}) \propto \pi_{good} \left(\prod_{j=1}^{d} (\beta_{good,j})^{x_j} \right) \tag{42}$$

We do the same for Bad Author, and classify $\mathbf{x}$ with whatever class maximized this probability.

Note that, had we taken a discriminative approach, we would have needed to invent weights $\mathbf{w}$ and w_0 and optimize for them. Having first told a probabilistic story, we were able to find closed forms for the parameters. Both discriminative and generative approaches have their own advantages.

NB: We are not limited to using MLE for the $\boldsymbol{\beta}_y$ and π_y probabilities above, but instead we can include priors and use MAP. We can use prior-likelihood conjugate pairs like Beta-Bernoulli for the class probability and Dirichlet-Multinomial for the class conditional feature probabilities. It turns out that using priors in this setting amounts to adding pseudo counts for certain classes or features, which can help with rare or unseen classes and features.

5.3.2 Gaussians + Continuous Features

One example of a generative classification model for continuous features are the Gaussian Generative Models from homework 2. We are still interested in:

$$p(y)p(\mathbf{x}|y) \tag{43}$$

Now, our conditionals are different Gaussians depending on the class:

$$p(\mathbf{x}|y=0) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0) \tag{44}$$

$$p(\mathbf{x}|y=1) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \tag{45}$$

and our priors on classes are

$$p(y=0) = \frac{N_0}{N}, \quad p(y=1) = \frac{N_1}{N} \tag{46}$$

where N_0 and N_1 are the number of training examples in classes 0 and 1 respectively. We make a prediction on a new observation by choosing the class that maximizes $p(y)p(\mathbf{x}|y)$.

6 Gradient Descent

Gradient descent is a general method for optimizing functions. In the context of machine learning, if we have parameter vector $\mathbf{w}$ and a loss function $\mathcal{L}$, we want to find the value of $\mathbf{w}$ such that $\mathcal{L}$ is minimized.

Formally, if we have model (e.g. logistic regression) $y = f(\mathbf{x}; \mathbf{w})$, data points $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with loss $\mathcal{L} = \sum_{i=1}^N \mathcal{L}(f(\mathbf{x}_i; \mathbf{w}), y_i)$, we perform a gradient update as

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}} \tag{47}$$

where η is the learning rate.

Stochastic gradient descent is more commonly used in practice. Instead of computing gradient of entire loss, we compute stochastic gradients:

1. Sample data point i, corresponding to $(\mathbf{x}_i, y_i)$
2. Compute loss and gradient for just that data point $\mathcal{L}^{(i)}(\mathbf{w})$
3. Update $\mathbf{w}$ based on this *stochastic gradient*

Similar to standard gradient descent, often more efficient in practice.

7 Perceptron

The perceptron is a simple linear model for binary classification. We have as before $h(\mathbf{x}; \mathbf{w}, w_0) = \mathbf{w}^\top \mathbf{x} + w_0$, where we classify an input $\mathbf{x}$ as positive if $h(\mathbf{x}) > 0$ and negative otherwise.

To train this model, we use the ReLU activation function:

$$f_{\text{RELU}}(x) = \max(x, 0) \tag{48}$$

The loss over our training set $(\mathbf{x}_i, y_i)$, where $y_i = +1$ for positive data points and $y_i = -1$ for negative, is:

$$\mathcal{L} = \sum_{i=1}^{N} f_{\text{RELU}}(-y_i h(\mathbf{x}_i)) \tag{49}$$

This loss allows us to take gradients easily: the stochastic gradient update is simply:

$$\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i \tag{50}$$

if y_i is incorrectly classified, and no update otherwise.
The perceptron is the simplest example of a neural network.

8 Neural Networks

Neural networks are complex models based on the idea of stacking multiple levels of computation on top of each other. One simple example of a neural network is *multi-layer perceptron* (MLP), which is just multiple perceptrons connected in a row. For example, for binary classification, we can use the following (2 layer) MLP:

$$h(\mathbf{x}; \mathbf{W}^1, \mathbf{w}_0^1, \mathbf{w}, w_0) = \mathbf{w}^\top f_{\text{RELU}}(\mathbf{W}^1\mathbf{x} + \mathbf{w}_0^1) + w_0 \tag{51}$$

Note that above, if $\mathbf{x} \in \mathbb{R}^d$, we have a matrix $\mathbf{W}^1 \in \mathbb{R}^{h \times d}$, where h is the number of *hidden units* (so called because they're neither the input of the network nor the output). Then $\mathbf{w} \in \mathbb{R}^h$. We can think of this as h individual units (the rows of $\mathbf{W}^1$) calculating their inner product with x, and then getting combined by $\mathbf{w}$.

More generally, we can think of neural networks as extremely flexible, tunable functions. For example, we can make a neural network that takes an input $\mathbf{x}$ (e.g. a document encoded to a vector somehow) to a vector of class probabilities, and use that for classification. Or we could take $\mathbf{x}$ (e.g. an image) and map it to a decimal, which could be a score.

8.1 Backpropagation

How do we adapt these functions? Well, after we've defined a loss function (which, in the case of classification can be negative log-likelihood, or in regression can be sum of squared error), we use the *backpropagation algorithm* to get the gradient of the loss for gradient descent. This is simply an application of the chain rule from calculus.

If we have a loss function $\mathcal{L}(h(\mathbf{x}), y)$ for a data point $(\mathbf{x}, y)$, as defined above (hereafter just abbreviated as $\mathcal{L}$), then we can think of our calculation in the forward direction as being the sequence of steps:

$$\begin{aligned}
\mathbf{g} &:= \mathbf{W}^1\mathbf{x} + \mathbf{w}_0^1 && (52)\\
\boldsymbol{\phi} &:= f_{\text{RELU}}(\mathbf{g}) && (53)\\
h &:= \mathbf{w}^\top\boldsymbol{\phi} + w_0 && (54)\\
\mathcal{L} &:= f_{\text{RELU}}(-y \cdot h) && (55)\\
&= -\mathbb{1}_{\{-y \cdot h > 0\}} y \cdot h && (56)
\end{aligned}$$

which we call the *forward pass*. Then, we can calculate the gradients with respect to each parameter by using those stored calculations:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial h}{\partial \mathbf{w}} \tag{57}$$

$$= \big(\mathbb{1}_{\{-yh>0\}} \cdot (-y)\big) \cdot \boldsymbol{\phi} \tag{58}$$

$$\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial h}{\partial w_0} \tag{59}$$

$$= \big(\mathbb{1}_{\{-yh>0\}} \cdot (-y)\big) \cdot (1) \tag{60}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_j^1} = \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial h}{\partial \boldsymbol{\phi}} \cdot \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{g}} \cdot \frac{\partial \mathbf{g}}{\partial \mathbf{w}_j^1} \tag{61}$$

$$= \big(\mathbb{1}_{\{-yh>0\}} \cdot (-y)\big) \cdot \mathbb{1}_{\{\boldsymbol{\phi}>0\}} \cdot \mathbf{w}^\top \cdot x_j I \quad \forall j = 1, \ldots, d \tag{62}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_0^1} = \frac{\partial \mathcal{L}}{\partial h} \cdot \frac{\partial h}{\partial \boldsymbol{\phi}} \cdot \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{g}} \cdot \frac{\partial \mathbf{g}}{\partial \mathbf{w}_0^1} \tag{63}$$

$$= \big(\mathbb{1}_{\{-yh>0\}} \cdot (-y)\big) \cdot \mathbb{1}_{\{\boldsymbol{\phi}>0\}} \cdot \mathbf{w}^\top \cdot I \tag{64}$$

where $\mathbf{w}_j^1$ is the jth row of $\mathbf{W}^1$.

The notation $\mathbb{1}_A$ (if you haven't seen it before) means 1 if A is true, otherwise 0. We call this the *backwards pass* because the order in which we calculate gradients is the last parameter first, then backwards through the model. Notice how in our expressions for the gradients, we have a lot of shared computation: in fact, the first term in each product is the same! That first term is an indicator on whether we got the prediction wrong, so if we got it right (i.e. $h \cdot y > 0$) then there's no gradient. The gradients in the first layer $(\mathbf{W}^1, \mathbf{w}_0^1)$ share the term $\partial h / \partial \mathbf{g}$ – the gradient of the second layer by the first.

The key trick is to store the intermediate calculations in the forward pass, and use those to calculate gradients in the backwards pass. In practice all of the above loops and indicator functions are rolled into matrix multiplications, so this efficient.

9 Activation Functions + Adaptive Basis Functions

If instead of using a ReLU activation function we had used a sigmoid function σ, then the last layer of our MLP above would have just become logistic regression, since then our neural network function is $\sigma(\mathbf{W}^1\mathbf{x} + \mathbf{w}_0^1)$. This is a very convenient endpoint for calculating multiclass probabilities (since that's what we use multiclass logistic regression for), but since backpropogation is a general technique, we can use any activation function to build our network. Some common functions are $\sigma, \tanh, \mathrm{ReLU}$, as well as many other variants.

There are a few ways to compare their behavior, though not much formal theory. For example, for the σ (sigmoid) activation function, it has the tendency to kill gradients when the input is too large or too negative, which makes it not very robust. The same is true for $\tanh$. However, for the ReLU activation, because the gradient is 0 whenever the input is negative, it's possible for the neuron to essentially ”die”, getting stuck at a negative value if the learning rate is too high, which can mean your network won't train at all.

What's adaptive about these basis functions is that the weights and bias are tuned through the training process. The final layer can treat the rest of the network as $\boldsymbol{\phi}(\mathbf{x})$, where $\boldsymbol{\phi}$ are basis functions adapted through training for the particular problem to be solved.

9.1 Shared Weights

Neural networks quickly reach a large number of parameters, especially as we add more and more layers. This can cause overfitting problems.

One solution is to have weights shared between different features ϕ_j. An example is the *convolution* operator, where our first transformation matrix might look something like

$$\mathbf{W}^1 = \begin{bmatrix} w_1 & w_2 & w_3 & 0 & \dots & & & 0 \\ 0 & w_1 & w_2 & w_3 & 0 & \dots & & 0 \\ 0 & 0 & w_1 & w_2 & w_3 & 0 & \dots & 0 \\ \vdots & & & & & & & \\ & \dots & & 0 & w_1 & w_2 & w_3 & 0 \\ & & \dots & & 0 & w_1 & w_2 & w_3 \end{bmatrix}$$

where w_1, w_2, w_3 are the only 3 weights.

10 Metrics

In binary classification we can sometimes frame our problem as predicting a yes/no or positive/negative result. This comes up in applications like banking fraud, anomaly detection, and medical diagnostics.

Let TP mean # of true positives, correctly classified 1's. Let TN mean # of true negatives, correctly classified 0's. Let FP mean # false positives, true 0's that were classified as 1's. Let FN mean # false negatives, true 1 s that were classified as 0 s. In what settings might you be more okay with false positives than false negatives?

Precision:

$$Precision = \frac{TP}{TP + FP} \tag{65}$$

True Positives Rate (also called Recall):

$$TPR = \frac{TP}{TP + FN} \tag{66}$$

False Positives Rate:

$$FPR = \frac{FP}{FP + TN} \tag{67}$$

ROC/AUC: Construct a graph by placing FPR on the x-axis and TPR on the y-axis. Holding constant a specific hyperparameter α, your model is a single point on this graph. As you let the parameter α range over a set of values, your model is represented by a set of points on this graph. By connecting the points into a curve (called ROC), we can evaluate the area underneath. This can be used to quantify the predictive power of a binary classification model. This is called the Area Under Curve (AUC).

Glossary

- Activation Function: A function that applies an element-wise non-linearity to its inputs. Generally does not have any parameters of its own. (9)
  - Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$
  - Rectified Linear Unit: $f(x) = \max\{0, x\}$
  - Tanh: $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$
- Basis Function: A basis function is a function $f : \mathbb{R}^b \to \mathbb{R}^d$. Generally, basis functions are used to map inputs into a new vector space where they have desirable properties (such as linearity for regression, or linear separation of classes for classification). One example of a basis function is a function in your code that extracts tags from a document, for example.
- Perceptron: A simple binary classification model that uses the sign of a linear function of the input as its label predictions.
- Regression: A model to calculate a function of the input data in a continuous space, e.g. a value in $\mathbb{R}^2$.
- Classification: A model to calculate a function of the input data that is in a discrete space, e.g. a color, or virus type.
- Optimization
  - Loss Function: In order to do any optimization, we have to have a way to measure the difference between the predictions from our model, and the true output values from the training data. The function that calculates this difference is called the Loss Function. Generally, in all optimization tasks, we want to minimize the value of the loss function.

  - Gradient: The vector of partial derivatives of a function with respect to some set of variables. The gradient vector points in the direction of greatest increase, and has a norm equal to the slope.
  - Gradient Descent: An algorithm for finding the argmin of a function by moving in the direction of steepest descent (i.e. opposite the gradient).

- Bayesian Inference
  - Likelihood: The likelihood of an observation **x** given a model parametrized by θ is written $p(\mathbf{x}|\theta)$. For understanding generative approaches, it's important to consider this as the extent to which our model could have generated our data.
  - Prior: A distribution that expresses one's beliefs about a random variable before any evidence is collected. Usually we try to use very wide, flat priors indicating that we don't know very much, but sometimes we use the prior to indicate that we know that a variable is inside some range: e.g. the prior distribution on a coin flip probability only has a support on $[0, 1]$ indicating that we know with certainty that it is inside that range.
  - Posterior: The posterior for model parameter θ represents our belief over the value of θ after we have seen our data D. It is written as $p(\theta|D)$.
    $p(\theta|D) \propto p(\mathbf{x}|\theta)p(\theta)$.
  - Bayes Rule: This is the central statistical property that we use all the time in this class.

    $$p(x, y) = p(x \mid y)p(y) = p(y \mid x)p(x) \implies \frac{p(y \mid x)}{p(y)} = \frac{p(x \mid y)}{p(x)}$$

    In practice for our purposes, we usually write it as $p(y|x) \propto p(x|y)p(y) = p(x, y)$. In other words, conditional probabilities are proportional to joint probabilities.
  - Posterior Predictive: a distribution over new data points that marginalizes (i.e. integrates over all possible values) out the parameter of interest.
  - MLE: The Maximum Likelihood Estimator for a parameter θ is the value of θ that maximizes the likelihood $p(D|\theta)$.
  - MAP: The MAP Estimator for a parameter θ is the value of θ that maximizes the posterior $p(\theta|D)$.

- Softmax: A function that takes a vector of scores or activations and maps them to a probability distribution (i.e. so that the sum of the values is 1):

$$\text{softmax}(\mathbf{z}) = \left[\frac{e^{z_1}}{\sum_{k=1}^{K} e^{z_k}}, \ldots, \frac{e^{z_K}}{\sum_{k=1}^{K} e^{z_k}}\right]$$

- Least Squares: Loss function for regression of the form

$$\mathcal{L} = \sum_{i=1}^{N} (h(\mathbf{x}_i) - y_i)^2$$

- Cross Validation: Breaking a dataset into k chunks for model selection, where for each chunk, we use $k-1$ of the chunks for training and the remaining for validation.
- Discriminative vs Generative Model: When trying to predict y from $\mathbf{x}$, we can either model the joint $p(y, \mathbf{x})$ (generative model) or only $p(y|\mathbf{x};\theta)$ (discriminative model) where θ are model parameters.