Machine Learning Notes

Ethan Dintzner and Hannah Ye

Contents

Chapter 2

Statistical Learning (Chapter 2) - Ethan's Notes

The essence of statistical learning is to estimate an output Y as a function of a set of predictors $X_1, ..., X_n := X$ such that $Y = f(X) + \varepsilon$, where ε is an error term that is normally distributed with a mean of zero. The main two applications are

1. **Predict** Y given some readily available inputs X as $\hat{Y} = \hat{f}(X)$, where $\hat{Y}$ is the predicted value of Y (assuming the error term is zero). The accuracy of $Y \to \hat{Y}$ depends on reducable and irreducable error: reducable error is something that can be decreased by statistical learning, and irreducable error is inherent random error (ε). Explicitly the error between Y and $\hat{Y} := E(Y - \hat{Y})^2$ can be written as

$$\begin{aligned} E(Y - \hat{Y})^2 &= E[f(X) + \varepsilon + \hat{f}(X)]^2 \\ &= [f(X) - \hat{f}(X)]^2 + \text{Var}(\varepsilon) \end{aligned}$$

   The term $[f(X) - \hat{f}(X)]^2$ is the thing minimized by statistical learning.

2. **Infer** relationships between the inputs and the outputs, but not necessarily need to predict them (lame). Look for the sign and magnitude of the relationship between different predictors and outputs.

The way you gotta do it is you have a set of data that you use to train your model, and then once it is trained it can predict Y fairly accurately for any input X. There are two types of learning methods that are discussed

1. **Parametric**. Step a) Assume that the function f is of an explicit form with parameters, for example assume that it is exponential form $f(x : a, k, b) = a \cdot e^{kx} + b$, linear form $f(x : a, b) = a \cdot x + b$ etc. Step b) need an algorithm that fits the form to the training data, fitting it to the set of parameters. The drawback is that you will likely not know the functional form of the phenomenon you observe.

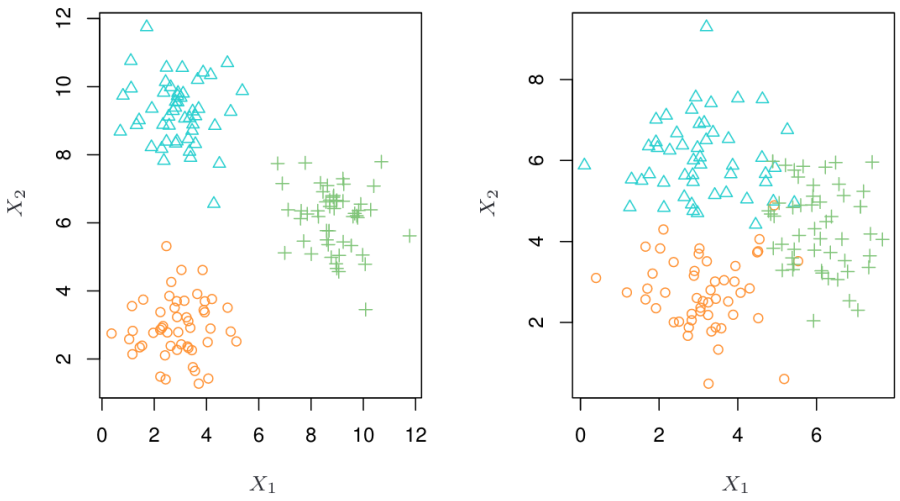
2. **Non-parametric**. You do not make assumptions about the functional form of f. The advantage is that you will be able to make predictions about really complicated functions with no *a priori* understanding of their form, but the disadvantage is that you need hella observations.

Why not use non-parametric / very flexible models? Because they are less interpretable than using non-flexible models. There are also two kinds of learning that are discussed: 1) Supervised and then 2) Unsupervised. The difference is that supervised learning you have a response variable Y that you can measure during learning / fitting, but unsupervised you have no response and you have to rely on organizing the inputs into something meaningful. One example is clustering, which is whether the measured inputs fall into meaningful categories. An example from the textbook is shown in the picture.

One side note from the text: use linear regression for quantitative variables and logistical for qualitative. However, most learning methods can be used for qualitative and quantitative variables.

To assess how well your model fits your data, the text lists the following:

Figure 1: Example of clustering data (unsupervised learning technique)



1. **Mean square error**. Most commonly used in regression. Defined as the following

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

The goal is to use a training method that gives the lowest test MSE, even better than a low training MSE. Once you have testing data you, you can compute the average squared prediction error for test observations (y_0, x_0)

$$Ave(y_0 - \hat{f}(x_0))^2$$

2. **Bias-Variance Trade-off**. The MSE can always be expressed in terms of the variance of $\hat{f}$, the bias of $\hat{f}$, and the error term ε:

$$E(y_0 - \hat{f}(x_0))^2 = Var(\hat{f}(x_0)) + [Bias(\hat{f}(x_0))]^2 + Var(\varepsilon)$$

The *variance* of $\hat{f}$ means the amount that $\hat{f}$ changes in response to different training data sets. The *bias* is the error that's associated with applying a mathematical model to a real life thing. **More flexible methods have more variance but less bias**.

3. **The Classification Setting**. The text basically says that if the training outputs are qualitative (e.g. identifying people's faces correctly), then you can count the amount of correct observations logically

$$\text{“error rate”} = \frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i)$$

The function I equals zero if correct and one if incorrect.

4. **The Bayes Classifier**. I didn't really understand this part, but it has something to do with assigning things to classes based on probability threshold.

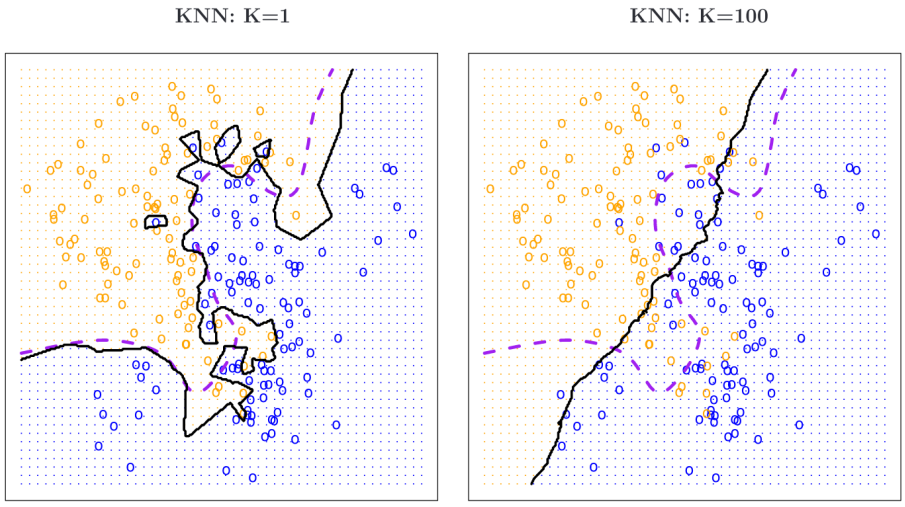
Figure 2: I thought this shit was saur funny - Ethan

5. **K-Nearest Neighbors**. Uses a subset of the training data $S = \{y_i : i \in \mathcal{N}_0\}, |S| = K$ close to a test data point to estimate the probability it is part of the same class j as the training points, given as the following:

$$Pr(Y = j : X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

You vary K for assigning probabilities to test points. Look at the figure for examples of $K = 1, K = 100$, really cool!

Figure 3: Example of K-nearest neighbors



That is all for the second chapter until the labs, which I will attempt at a later date.

Statistical Learning (Chapter 2) - Hannah's Notes

2.2 Assessing Model Accuracy

1. Quality of fit

   MSE: we want estimate $\hat{f}(x_0)$ to be approximately equal to y_0, where y_0 is *previously unseen test observation.* As flexibility of stat learning method increases, monotonic decr. in training MSE and U-shape in test MSE (lead to overfit).

   Bias-Variance Trade Off: U-shape in test MSE curves results from this. Expected test MSE for given x_0 can always be decomposed into the sum of 3 fundamental quantities: var of $\hat{f}(x_0)$, squared *bias* of $\hat{f}(x_0)$, and var of error terms ε.

   $$E(y_0 - \hat{f}(x_0))^2 = Var(\hat{f}(x_0)) + [Bias(\hat{f}(x_0)))]^2 + Var(\varepsilon)$$

   first term = expected test MSE at x_0, refers to average test MSE from repeated estimated f with large numbers of training sets at x_0. Equation tells us that we need *low var* and *low bias.* Variance: amount by which $\hat{f}$ (predicting function) would change if we estimated it using a different training data set, and bias is the error that is introduced by approximating a real-life problem by a simpler model. General rule is that more flexible method $\rightarrow$ increase in variance and decrease in bias: if Δbias greater than that of variance, than expected test MSE declines and vv.

   The Classification Setting Quantify accuracy of estimate $\hat{f}$ is the training *error rate* (proportion of mistakes made if we apply to training observations) Bayes Classifier: test error rate is minimized on avg by a classifier that *assigns each observation to the most likely class, given its predictor values.* Assign test observation with predictor vector x_0 to the class j for which

   $$Pr(Y = j|X = x_0)$$

   is largest (probability that Y = j, given observed predictor vector x_0. Produces .. something

   K-Nearest Neighbors With Bayes, we don't know the conditional distribution of Y given X for real data. KNN tries to estimate this and then classify given obs to class with highest *estimated* prob. Given pos int K and test obs x_0, classifier first identifies K points in training data closest to x_0, rep by $\mathcal{N}_0$ – then est cond prob for class j as fract of points in $\mathcal{N}_0$ whose response vals = j:

   $$Pr(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

   and classifies test obs x_0 to class with greatest prob from above. Low K $\rightarrow$ more flexible and decrease in training error rate (at first), and vv.

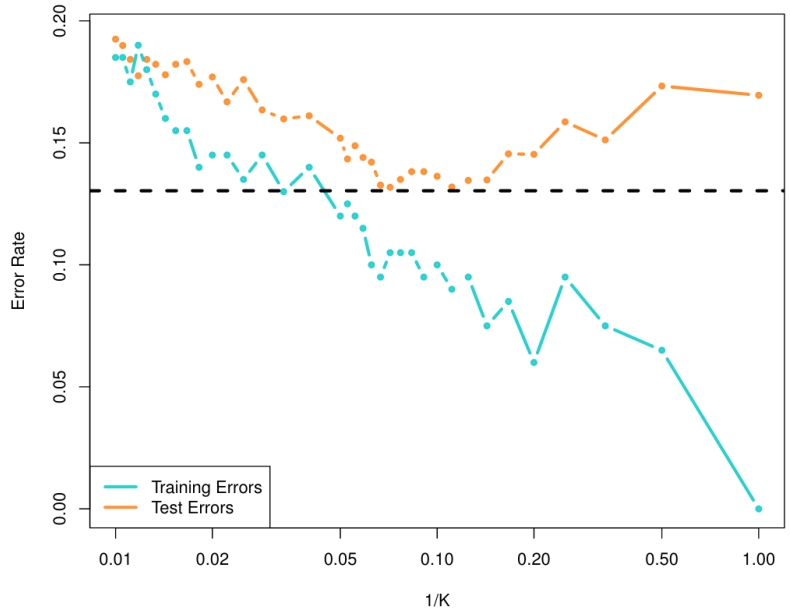
2.3 Intro to Python
At this point, I am supposed to practice. ahahaha

Chapter 2 coding exercises

These were made by Ethan and Hannah to put us through coding pains :)

1. Use the advertisement dataset to compute the MSE for simple linear regression, for all three variables.
2. Train the advertisement data (linear regression) with half of the data, and then calculate the test MSE with the other half.
3. Use linear regression on half of the cars data to "train" a model (mpg vs. cylinders) and give a discrete outout on the expected amount of cylinders.
4. Calculate the training error and then use the other half of the data set to calculate the test error.
5. KNN exercise: use the college dataset to plot a histogram of dropout rate (x-axis), room and board (y-axis), and then top 25% college (color indicator).
6. For different K values, test a 2D array of evenly spaced points and make a decision line based on when the probability of one state is over 0.5 (reproduce the figure above from the textbook)
7. Split the data into two random halves, do KNN for different K values, and plot the error rate dependence on K. Use the boundaries from the training data to predict test data categories, and replicate the figure below from the textbook

Figure 4: Figure for exercise 7



Chapter 3

Statistical Learning (Chapter 3) - Ethan's Notes

Linear regression is useful to learn because many of the more complicated learning methods are actually generalizations of linear regressions.

Simple Linear Regression
Assumes that there is an approximately linear relationship between a predictor X and a response variable Y, where "regressing Y on X" is written as

$$Y \approx \beta_0 + \beta_1 X$$

A model for this after training with observation points (data in the form of $\{x_i, y_i\}_{1 \leq i \leq n}$) is formally written as

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

To calculate the residuals e_i of the linear model generated from the training data, you can proceed by computing $e_i = y_i - \hat{y}_i$. Using this you can calculate the residual sum of squares

$$RSS = e_1^2 + e_2^2 + ... + e_n^2$$

And with some handwavy math from the textbook ("using some calculus") the least square coefficient estimates are defined as $\hat{\beta}_0 = \bar{y} + \hat{\beta}_1 \bar{x}$, with $\bar{x}, \bar{y}$ as sample means. The issue is that the sample means can be biased, or at least slightly different from the population means, which makes the model regression line slightly different than the real thing. You can calculate the square standard error of the sample mean by the following

$$SE(\hat{\mu})^2 = \frac{\sigma^2}{n}$$

The important observation is that the error decreases with the size of the sample. Assuming that the variance have a common variance σ^2, then you can compute the square standard error of the regression coefficients

$$SE(\hat{\beta}_0)^2 = \sigma^2 [\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^{n} (x_i \bar{x})^2}]$$
$$SE(\hat{\beta}_1)^2 = \frac{\sigma^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

You can calculate the 95% confidence interval by using the standard error formulas

$$\hat{\beta}_1 \pm 2 \cdot SE(\hat{\beta}_1)$$

Hence, the interval $\left[\hat{\beta}_1 - 2 \cdot SE(\hat{\beta}_1), \hat{\beta}_1 + 2 \cdot SE(\hat{\beta}_1)\right]$ contains the population variable. Also the t statistic is defined as $t = \frac{\hat{\beta}_1 - 0}{SE(\hat{\beta}_1)}$, and usually when it is less than 5% or 1% it means that the trend is significant.

Simple Linear Regression Model Validation
Can use the RSS which was previously defined. Can use R^2, which is computed by

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

The R^2 value is actually equal to the correlation squared, so they are interchangeable.

Multiple Linear Regression
Multiple linear regression is important if you have multiple predictors for a response variable. It takes the general form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p + \varepsilon$$

If you have the estimated regression coefficients (i.e., the coefficients that minimize the error between the data and the model) as $\{\hat{\beta}_i\}_{i \in I}$, then the general form of the RSS is the same as in the simple case, but $\hat{y}$ is computed with the general form of linear regression. To reject the null hypothesis that the coefficients are non-zero, you can compute the F-statistic and want to show that it is greater than 1

$$F = \frac{\left[\sum(y_i - \bar{y}) - \sum(y_i - \hat{y})\right]/p}{\left(\sum(y_i - \hat{y})\right)/(n - p - 1)}$$

Remark. *If the predictors are all close to zero, then the denomenator of the F-statistic is close to σ^2.*

If you have a set of p predictors of a response, then why not just try all of the possible models? The reason is from combinatorics: there are 2^p possible models. The text discusses a few methods for model choice

1. **Forward selection.** This method is kind of like a greedy algorithm. You start with a model that just contains an intercept, and then you add variables one by one that result in the lowest RSS. This is continued until some stopping rule is satisfied.
2. **Backward selection.** This method is not as greedy, but starts with all the variables in the model. It then removes the one with the highest p-value and continues until a stopping criteria is reached.
3. **Mixed selection.** Add variables one by that provide the best fit. But, while adding the variables, if any of the others rise above a certain threshold, then remove them. Continue this cycle until the model is created.

The residual standard error (RSE) is a way of measuring model fit

$$RSE = \sqrt{\frac{1}{n - p - 1} RSS}$$

Regression with qualitative (indicative) data For the simple case, if a predictor is binary

$$x_i = \begin{cases} 1 & x_i \text{ true} \\ 0 & x_i \text{ false} \end{cases}$$

Then the associated model is

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i = \begin{cases} \beta_0 + \beta_1 + \varepsilon_i & x_i \text{ true} \\ \beta_0 + \varepsilon_i & x_i \text{ false} \end{cases}$$

You could also code it as a 1/-1 scheme, but the only thing that would change is the interpretation of the coefficients. If you have multiple qualitative states of a variable x_{1_i}, x_{2_i} as

$$x_{1_i} = \begin{cases} 1 & x_{1_i} \text{ true} \\ 0 & x_{1_i} \text{ false} \end{cases}$$
$$x_{2_i} = \begin{cases} 1 & x_{2_i} \text{ true} \\ 0 & x_{2_i} \text{ false} \end{cases}$$

with the associated model

$$y_i = \beta_0 + \beta_1 x_{1_i} + \beta_2 x_{2_i} + \varepsilon_i = \begin{cases} \beta_0 + \beta_1 + \varepsilon_i & x_{1_i} \text{ true}, x_{2_i} \text{ false} \\ \beta_0 + \beta_2 + \varepsilon_i & x_{1_i} \text{ false}, x_{2_i} \text{ true} \\ \beta_0 + \varepsilon_i & x_{1_i} \text{ false}, x_{2_i} \text{ false} \end{cases}$$

Interaction Terms

If you have a simple linear model $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \varepsilon$, and the predictors X_1 and X_2 have an effect on each other, then the model can be rewritten with X_1, X_2 as a product of each other and another free parameter

$$\begin{aligned} Y &= \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \varepsilon \\ &= \beta_0 + (\beta_1 + \beta_3 X_2) X_1 + \beta_2 X_2 \\ &= \beta_0 + \widetilde{\beta_1} X_1 + \beta_2 X_2 \qquad (\widetilde{\beta_1} := \beta_1 + \beta_3 X_2) \end{aligned}$$

In a linear regression model where X_1 and X_2 are interacting, you must include $X_1, X_2,$ as well as $X_1 \times X_2$ or else the interaction term will mean something completely different, and especially since X_1, X_2 are correlated with their interaction term. It is good practice.

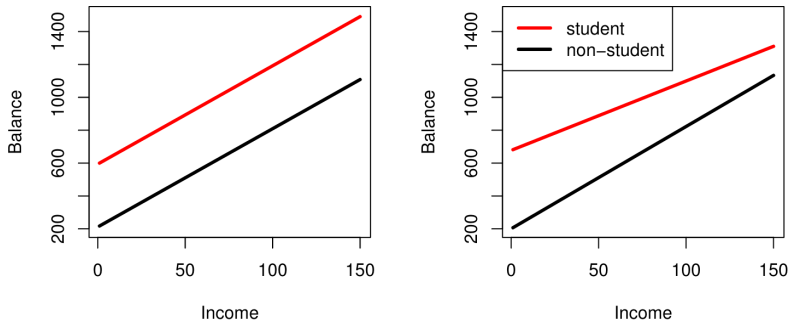
Interaction terms also work with binary / qualitative models. If you have a linear model of two variables X_1, X_2, with X_1 continuous and X_2 as a binary variables, then the linear model is

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases}$$

If there is interaction between X_1 and X_2, then you can rewrite

$$\begin{aligned} Y &= \beta_0 + \beta_1 X_1 + \beta_2 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases} + \beta_3 X_1 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases} \\ &= \beta_0 + \left(\beta_1 + \beta_3 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases} \right) X_1 + \beta_2 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases} \\ &= \beta_0 + \widetilde{\beta_1} X_1 + \beta_2 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases} \qquad \left(\widetilde{\beta_1} := \beta_1 + \beta_3 \begin{cases} 1 & X_2 \text{ true} \\ 0 & X_2 \text{ false} \end{cases} \right) \end{aligned}$$

Figure 8: Plot of qualitative linear models with and without interaction terms. The idea is that including the interaction term results in a different slope.



Polynomial Regression
For the one-predictor case, the general formula for polynomial regression is

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + ... + \beta_n X^n$$

The final part of this chapter talks about limitations to linear regression. They are enumerated below

1. **Non-linearity of data.** When the data does not follow a linear trend, then it is probabily appropriate to follow a different technique, because this means high model bias. A good thing to plot is the residuals versus the predictor (x-axis: X_i; y-axis: $e_i = y_i - \hat{y}_i$)

2. **Correlation of error terms.** If the error terms ε_i are correlated, then the sense of confidence in the model will be overestimated. For instance, it will make the 95% confidence intervals smaller than they actually are. Time series data typically tend to have positively correlated errors (time points taken close to each other have similar sign). This is called *tracking*.

3. **Non-constant variance of error terms.** This means when your residuals make a shape that is non-constant, like a funnel for example. This can be fixed by applying transforms (e.g. a log transform) to your data and verifying by replotting the residuals.

4. **Outliers.** Points in a dataset that have really high residuals. A good way to catch outlies is to plot leverage vs. *studentized residuals*. The datapoints that have the most weight on the model and the biggest errors are the outliers typically. I found this stack exchange post that has code for getting these values.

5. **High leverage points.** These points that where the predictors are far from the normal range of observations, as opposed to normal outliers where maybe just the residuals are high. For simple linear regression, the *leverage statistic* is calculated by

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{i'=1}^{n}(x_{i'} - \bar{x})^2}$$

   For more complicated linear models, you can use an sklearn package to find the diagonal elements of the H matrix.

6. **Colinearity of predictors.** If two predictors are extremely highly correlated, then it could cause issues with the models, especially their regression coefficients. This is because changing the model

only slightly will cause drastic changes in the coefficient estimates to the colinear predictors, making the model less meaningful.

A way to test colinearity is with the *variance inflation factor*. If you have n predictors, the way you compute the VIF for all of them is by regressing all of the predictors against each other

$$\begin{aligned} X_1 &= \alpha_0 + \alpha_2 X_2 + \alpha_3 X_3 + ... + \alpha_n X_n \\ X_2 &= \alpha_0 + \alpha_1 X_1 + \alpha_3 X_3 + ... + \alpha_n X_n \\ &... \\ X_n &= \alpha_0 + \alpha_1 X_1 + \alpha_2 X_2 + ... + \alpha_{n-1} X_{n-1} \end{aligned}$$

And then you compute the reciprocal of the complement for each model's R^2

$$\begin{aligned} \text{VIF}_1 &= \frac{1}{1 - R_1^2} \\ \text{VIF}_2 &= \frac{1}{1 - R_2^2} \\ &... \\ \text{VIF}_n &= \frac{1}{1 - R_n^2} \end{aligned}$$

The final part of this chapter is a comparison of KNN with linear regression. The way you can use KNN for best fit curves is really cool: Let K be the number of nearest neighbors to an observation x_0. Let $\mathcal{N}_0$ be the set of K neighbors closest to x_0. Then, $\hat{f}(x_0)$ is modeled as

$$\hat{f}(x_0) = \frac{1}{K} \sum_{x_i \in \mathcal{N}_0} y_i$$

The textbook says that KNN is much better than linear regression when the relationship is non-linear, but linear regression outperforms when it is linear. KNN falls behind whenever you add *noise predictors* though, or predictors that don't have to do with the response. So, you could say that KNN requires more precise model choice. Also, KNN falls risk to the *curse of dimensions*, where data in higher dimensions becomes further apart (so observations lack close-by data).

Statistical Learning (Chapter 3) - Hannah's Notes

3.1 Simple Linear Regression

Formular:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$$

where $\hat{y}$ indicates prediction of Y on basis of X, and the B terms are estimated terms.

$$e_i = y_i - \hat{y}_i$$

represents the ith residual, and the residual sum of squares (RSS) is:

$$RSS = (e_1)^2 + (e_2)^2 + ... + (e_n)^2$$

expanded into:

$$RSS = (y_1 - \hat{\beta}_0 - \hat{\beta}_1 x_1)^2 + (y_2 - \hat{\beta}_0 - \hat{\beta}_2 x_2)^2 + ... + (y_n - \hat{\beta}_0 - \hat{\beta}_n x_n)^2$$

In order to minimize RSS, choose $\hat{\beta}_0$ and $\hat{\beta}_1 x_1$. See textbook for the equations.
Assessing the Accuracy of the Coefficient Estimates
Generally use standard error of the computed mean ($\hat{\mu}$):

$$var(\hat{\mu}) = SE(\hat{\mu})^2 = \frac{\sigma^2}{n}$$

Tells us the average amount that the estimate differs from the actual value of μ, and that deviation shrinks with n. Can use similar method for estimating accuracy of $\hat{\beta}_0$ and $\hat{\beta}_1$ (see textbook). Generally don't know standard deviation, but can estimmate it with the residual standard error ($\hat{SE\beta}_1$)

Multiple Linear Regression

Model takes the form:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_p X_p + \varepsilon$$

Estimating the Regression Coefficients
Same method as simple linear regression (minimize sum of squared residuals when choosing parameter values).

1. *Is at least one of the predictors X_1, X_2, .. X_p useful in predicting the response?*
   Null hypothesis is that all the parameters equal zero, and the alternative is that at least one is non-zero. Test this by computing the F-statistic:

   $$F = \frac{(TSS - RSS)/p}{RSS/(n-p-1)}$$

   where TSS is $\sum (y_i - \bar{y})^2$ and RSS is $\sum (y_i - \hat{y})^2$. When there is no relationship between response and predictors, F stat ≈ 1 (denominator and numerator both equal variance). TO reject the null hypothesis, we need a large F-stat is n is small (can compute necessary p-value given n and p). Cannot just look at the p-values for each variable, since if number of predictors is large the possibility that p-values are 'significant' increases. F-statistic won't do this, as it adjusts for the number of predictors.

2. *Deciding on Important Variables*
   We can use some approaches to choosing a smaller set of models (rather than all possible ones for all variables) to consider when trying to figure out which predictors are associated with the response.

Forward selection
Begin with null model (has intercept but no predictors). Fit p simple linear regressions and put lowest RSS var into model, then fit p two-var linreg ... Continue until some stopping rule is satisfied.

Backward selection
Remove variable with the largest p-val, and repeat until some threshold of p-values is reached.

Mixed selection
Continually add variables that provide the best fit, and if at any point the p-value for one of the variables ¿ threshold, remove variable. Continue this until: All vars have a sufficiently low p-value, and all those not in model would have too large of a p-value.

3. *Model Fit*
Two most common = RSE and R^2 (square of correlation of the response and the variable in simple regression, square of correlation between response and fitted linear model) Can also be useful to plot the data ... if non-linear residuals, could suggest *synergy* or *interaction* (think cooperativity).

4. *Predictions*
Once we've fit our multiple regression model, we can predict response .. but with 3 kinds of uncertainty:

   (a) Coeff estimates incorporated into the *least squares plane*:

   $$\hat{Y} = \hat{\beta}_0 + b\hat{e}ta_1X_1 + ... + \hat{\beta}_pX_p$$

   Which is only an estimate for the *true population regression plane* Can compute a confidence interval to determine how close $\hat{Y}$

   (b) additional source of error = *model bias* (textbook says to ignore)

   (c) even if we knew trew values of parameters, we can't predict response value perfectly because of random error ε (irreducible)

   Confidence intervals -¿ quantifying the uncertainty surrounding the average. Prediction intervals -¿ quantifying uncertainty surrounding a particular group / sample.

Other considerations in the Regression Model

Qualitative Predictors

Predictors with Only Two Levels
If qualitative predictor = binary, create a dummy variable that takes on 2 numerical values:

$$x_i = \begin{cases} 1 & i\text{th person owns a house} \\ 0 & i\text{th person does not own a house} \end{cases}$$

Results in model:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i = \begin{cases} \beta_0 + \beta_1 + \varepsilon_i & \text{if} i\text{th person owns a house} \\ \beta_0 + \varepsilon_i \text{if} i\text{th person does not} \end{cases}$$

Note that the values you choose to code alter interpretation of the coefficients.

Qualitative Predictors with More than 2 Levels
Can create additional binary dummies such that:

$$x_i1 = \begin{cases} 1 & i\text{th person is from the South} \\ 0 & i\text{th person is not from the South} \end{cases}$$

combined with:

$$x_i2 = \begin{cases} 1 & i\text{th person is from the West} \\ 0 & i\text{th person is not from the West} \end{cases}$$

can be used in the regression equation in order to obtain the model:

$$y_i = \beta_0 + \beta_1 x_i1 + \beta_2 x_i2 + \varepsilon_i = \begin{cases} \beta_0 + \beta_1 + \varepsilon_i & i\text{th person is from the South} \\ \beta_0 + \beta_2 + \varepsilon_i & i\text{th person is from the West} \\ \beta_0 + \varepsilon_i & i\text{th person is from the East} \end{cases}$$

Level with no dummy variable (East in example) is *baseline*.

Extensions of the Linear Model
We assume that the relationship between the predictors and the response are *additive* and *linear*. Additive -¿ association between predictor and response values does not depend on the values of the other predictors. Linearity -¿ change in response associated with a one-unit change in predictor is constant, regardless of the value of the predictor. How do we extend the model past these assumptions?

Removing the Additive Assumption
Consider possibility of interaction effect (i.e. cooperativity). Standard linear regression model with two var:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$$

Here, regardless of the value of X_2, 1-unit increase in X_1 -¿ β_1 unit increase in Y. To extend model to include interaction effect, can include third predictor (*interaction term*):

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \varepsilon$$

This can be rewritten as:

$$Y = \beta_0 + (\beta_1 + \beta_3 X_2) X_1 + \beta_2 X_2 + \varepsilon$$

Let the parenthetical shit be $\widetilde{\beta_1}$ -¿ 2 will lead to change in association between X_1 and Y, and vv. Note about p-values: sometimes interaction term has small p-val, but associate effects do not. *Hierarchial principle* states that if we include interaction -¿ include main effects even if p-vals assoc with coeffs $\neq$ significant. Can add interaction terms for qualitative variables too (see textbook, I got lazy).

Non-linear Relationships
Can use polynomial regression. If shape suggests that the points may have (for e.x.) a quadratic form:

$$mpg = \beta_0 + \beta_1(horsepower) + \beta_2(horsepower^2) + \varepsilon$$

Can make into the form of a multiple linear regression model where X_1 is horsepower and X_2 is $horsepower^2$.

Potential Problems

(a) Non-linearity of the data
Residual plots = useful graphical tool to see whether your data is nonlinear.

(b) Correlation of Error Terms
Linear regression model assumes that the error terms ($\varepsilon_1, \varepsilon_2, ...$) are uncorrelated. If they are actually correlated, estimated SEs will tend to be $\geq$ than actual errors, and residuals will tend to track with their neighbors. This happens frequently in time series data (obs at adjacent time points -¿ pos corr errors).

(c) Non-constant Variance of Error Terms
If there's non-constant variance (heteroscedasticity), can identify from funnel shape in residuals plot. Soln: transform the response variable (log or sqrt Y) or fit the model by *weighted least squares* with weights proportional to the inverse variances.

(d) Outliers
Can clearly identify them using residual plots, and can plot studentized residuals:

$$se_i = \frac{e_i}{SE}$$

Observations with studentized residuals ≥ 3 = possible outliers.

(e) High Leverage Points
Kind of the opposite of outliers ... observations with high leverage have a unusual predictor value. Easier to identify in simple linear regression, but in multiple we can have an observation that is within range of individual predictor values but unusual in terms of full set. Can quantify by computing *leverage statistic* (large value = high leverage). For simple linear regression:

$$h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum}$$

Chapter 3 coding exercises

1. For a simple regression and a two variable regression, plot the regression line / plane and the residuals mapped onto it. Calculate the R^2 values too. Use the sales dataset

2. For the schemes above, make residual plots for each predictor (x_i vs $y_i - \hat{y}_i$).

3. Use the college dataset and make an interesting multivariate regression. Calculate the F-statistic. Try to do another multivariate regression with an interaction term. Make residual plots for each variable, as well as an overall residual plot.

4. Implement the following for the colleges dataset, with the stopping criteria something like having a model with 8 predictors.

   (a) Forward selection.

   (b) Backward selection.

   (c) If it the above easy, try mixed selection with a p-value threshold of 0.1 and then if it terminates with more than 8 variables then do backward selection.

(d) Calculate the F associated with (a-c).

(e) Make residual plots for everything.

5. For one of the models above, make a *leverage vs. studentized residuals* plot. Use the linked code in Ethan's notes.

6. Choose one of the multivariate models above and test for colinearity of predictors with VIF. Make a bar graph of VIF for each predictor.

7. Implement and plot KNN for a maybe two-predictor dataset, and then try it for an n-predictor.

Chapter 4

Statistical Learning (Chapter 4) - Ethan's Notes

Classification
Linear regression does not work if the response variable is not quantitative. This is where classification comes in: the response variable for classification methods is qualitative or categorical. Why not use linear regression for classification? Because if you had predictors and a response that looked something like this

$$Y = \begin{cases} 1 & \text{if tree} \\ 2 & \text{if tiger} \\ 3 & \text{if bacteria} \end{cases}$$

Then the way you order the outputs makes a linear model completely meaningless because $1 \to 2 \to 3$ juts makes no sense. If there are two classes (0,1), it is a little bit better, but some estimates might be below or above 0 or 1 which would make it hard to interpret as probabilities.

Logistic Regression
Logistic regression models a probability based on predictors. Its general form for a single predictor (X) of the probability that Y is true is

$$\Pr(Y = 1|X)$$

The way that it works in practice is described in the next subsections.

1. **The logistic model.** For instance, if you want a function that returns the probability of the model above, you need it to map to values between 0 and 1. Its functional form is

   $$Y = p(X) := \frac{\exp(\beta_0 + \beta_1 X)}{1 + \exp(\beta_0 + \beta_1 X)}$$

   With some manipulation, you get the term called the *odds*

   $$\frac{p(X)}{1 - p(X)} = \exp(\beta_0 + \beta_1 X)$$

   The odds are basically the chance that it will occur over the chance that it won't occur. If $p(X) = 0.2$, that means it is a 20% chance that it will occur, but the *odds* are $\frac{\text{occurs}}{\text{doesn't occur}} = 0.25$. If you take the log of both sides

   $$\ln\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X$$

   The LHS is called the *logit*, and the RHS is called "linear in X" in this situation.

2. **Estimating the regression coefficients.** The idea is that you have to fit a likelihood function that isn't explained too much in the textbook

   $$l(\beta_0, \beta_1) = \prod_{i=1}^{n} p(x_i)^{y_i} (1 - p(x_i))^{1-y_i}$$

   The textbook says you should just use packages to do this. The z-statistic is how one should evaluate the null hypothesis that the response variable does not depend on the predictor X.

3. **Making predictions from trained logistic model.** The formal model from a logistic regression of two dimensions with estimated coefficients $\hat{\beta}_0, \hat{\beta}_1$ is

$$\hat{p}(X) = \frac{\exp\left(\hat{\beta}_0 + \hat{\beta}_1 X\right)}{1 + \exp\left(\hat{\beta}_0 + \hat{\beta}_1 X\right)}$$

In logistic regression, qualitative predictors can also be used.

4. **Multiple logistic regression.** The general form of logistic regression follows the form

$$p(X) = \frac{\exp\left(\beta_0 + \beta_1 X_1 + ... + \beta_k X_k\right)}{1 + \exp\left(\beta_0 + \beta_1 X + ... + \beta_k X_k\right)}$$

To estimate the coefficients, you optimize a similar likelihood function as mentioned in section 2.

5. **Multinomial logistic regression.** Above, we have seen ways to obtain logistic regression models when the response variable is binary (0,1). What if the response variable has multiple classes, like tree, bacteria, or tiger??? Denote the number of classes of a logistic model as K, and let p be the number of predictors (as a vector $\mathbf{X} = x_1, ..., x_p$). Then, for each category $1, 2, ..., i, ..., K-1, K$ we can write the associated logistic regression models

$$\begin{aligned}
\Pr(Y = 1 : \mathbf{X}) &= \frac{\exp\left(\beta_{1,0} + \beta_{1,1} x_1 + ... + \beta_{1,p} x_p\right)}{1 + H} \\
\Pr(Y = 2 : \mathbf{X}) &= \frac{\exp\left(\beta_{2,0} + \beta_{2,1} x_1 + ... + \beta_{2,p} x_p\right)}{1 + H} \\
&... \\
\Pr(Y = i : \mathbf{X}) &= \frac{\exp\left(\beta_{i,0} + \beta_{i,1} x_1 + ... + \beta_{i,p} x_p\right)}{1 + H} \\
&... \\
\Pr(Y = K - 1 : \mathbf{X}) &= \frac{\exp\left(\beta_{K-1,0} + \beta_{K-1,1} x_1 + ... + \beta_{K-1,p} x_p\right)}{1 + H} \\
\Pr(Y = K : \mathbf{X}) &= \frac{1}{1 + H}
\end{aligned}$$

with $H := \sum_{l=1}^{K-1} \exp\left(\beta_{l,0} + \beta_{l,1} x_1 + ... + \beta_{l,p} x_p\right)$. Rearranging similar to section 1, we have for $i \in [1, K-1]$ that

$$\log\left(\frac{\Pr(Y = i : \mathbf{X})}{\Pr(Y = K : \mathbf{X})}\right) = \beta_{i,0} + \beta_{i,1} x_1 + ... + \beta_{i,p} x_p$$

What exactly does this mean? It means that if you choose the case K as a "baseline", then if some predictor $x_j, j \in [1, p]$ increases by one unit then the odds that the set of predictors is associated with the class i increases by $\exp(\beta_{i,j})$. In other words, if x_j increases by one unit then

$$\frac{\Pr(Y = i : \mathbf{X})}{\Pr(Y = K : \mathbf{X})}$$

increases by $\exp(\beta_{i,j})$. Some logistic regression approaches assume that there is no such "baseline" and regress every category with the same formula as the i^{th} category in the equations above. Then, the odds between any categories i, j can be expressed as

$$\frac{\Pr(Y = i : \mathbf{X})}{\Pr(Y = j : \mathbf{X})} = (\beta_{i,0} - \beta_{j,0}) + (\beta_{i,1} - \beta j, 1) x_1 + ... + (\beta_{i,p} - \beta_{j,p}) x_p$$

Generative models for classification
This is basically just another method to get estimations for $\Pr(Y = i : \mathbf{X})$. Why would we want to do that? It is because this method is better for smaller, normally distributed sample size, if classes are much different, and easily generalizable to multiple classes.

Let π_i be the chance that a randomly chosen observation falls into the category i (the prior probability), and $f_i(x)$ be the density function of $\mathbf{X}$ for an observation that comes from the i^{th} class. Then,

$$\Pr(Y = i : \mathbf{X}) = \frac{\pi_i f_i(x)}{\sum_{l=1}^{K} \pi_l f_l(x)}$$

This is referred to as the *posterior* probability p_i. The issue though: how do you estimate the density function?

1. **Linear discriminant analysis for one predictor, K classifications.** This method assumes that density function is Gaussian, i.e. follows the form

$$p_i(x) = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{1}{2\sigma_i^2}(x - \mu_i)^2\right)$$

   If you take a log of each of these probability equations, and assume that there is a shared variance term σ across all classes, you end up with a set of K Bayes classifier functions $\delta_1, \delta_2, ..., \delta_i, ..., \delta_K$ where

$$\delta_i(x) = x \cdot \frac{\mu_i}{\sigma^2} - \frac{\mu_i^2}{2\sigma^2} + \log \pi_i$$

   At this point, whichever function has the highest value is associated with which class a test observation is most likely part of. The decision boundary between any two classes is $x = \frac{\mu_i + \mu_j}{2}$. In practice, these parameters have to be estimated (and only if we are sure that each of the classifiers are normally distributed). They are calculated as such

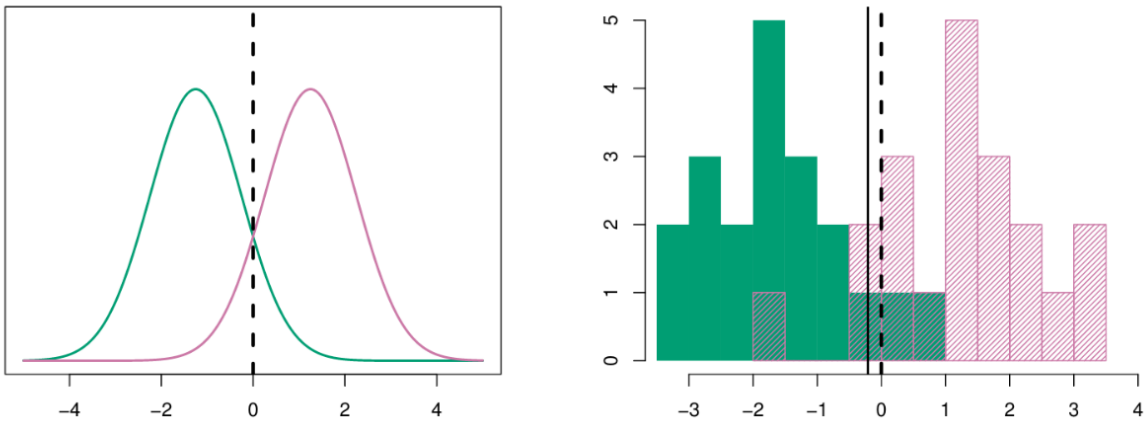
$$\begin{aligned}
\hat{\mu_i} &= \frac{1}{n_i} \sum_{j:y_j=i} x_j \\
\hat{\sigma}^2 &= \frac{1}{n - K} \sum_{k=1}^{K} \sum_{j:y_j=i} (x_j - \hat{\mu}_k)^2 \\
\hat{\pi}_i &= \frac{n_i}{n} \\
\implies \hat{\delta}_i(x) &= x \cdot \frac{\hat{\mu}_i}{\hat{\sigma}^2} - \frac{\hat{\mu}_i^2}{2\hat{\sigma}^2} + \log(\hat{\pi}_i)
\end{aligned}$$

   The process for finding each Bayes classifier is by computing each of the above parameters in order, and then plugging them into $\hat{\delta}_i$. The issue with this is that it is very stringent, as each class is not allowed to have its own specific variance σ.

2. **LDA for more than one predictor.** This assumes that you have a vector $X = (x_1, ..., x_p)$, your sample has a mean of $\mu = (\mu_1, ..., \mu_p)$, and a covariate matrix called $\mathbf{\Sigma}$, then the distribution of a classifier is

$$f(X) = \frac{1}{\sqrt{(2\pi)^p \det \mathbf{\Sigma}}} \exp\left(-\frac{1}{2}(X - \mu)^T \mathbf{\Sigma}^{-1}(x - \mu)\right)$$

Figure 9: Application of LDA for one predictor and two classifications.



Using similar manipulation to the previous section, we can find that the probability of an observation being in the ith class is (with μ_i as the class-specific mean vector, π_i is the number of samples in class i over the total number of samples)

$$\delta_i(x) = X^T \mathbf{\Sigma}^{-1} \mu_i - \frac{1}{2} \mu_i^T + \log \pi_i$$

for $i \in [1, K]$ classifiers. The issues with this are the assumptions of a common covariance matrix, which may not be true.

3. **QDA for more than one predictor.** QDA is like LDA except for it has the advantage of being much more flexible because it allows each class to have its own covariance matrix. Basically use this over LDA unless you have a very small sample size. Obviously calculate each δ_i and then choose the maximum for what category something is in.

$$\delta_i = -\frac{1}{2} X^T \mathbf{\Sigma}_i^{-1} X + X^T \mathbf{\Sigma}_i^{-1} \mu_i - \frac{1}{2} \mu_i^T \mathbf{\Sigma}_i^{-1} \mu_i - \frac{1}{2} \log \det \mathbf{\Sigma}_k + \log \pi_i$$

4. **Naive Bayes.** It has only one assumption, that the predictors are independent of each other. The math for this was a little hairy so I will probably revisit this section if I need to do such analysis.

Next, the text does an empirical comparison of the different classification methods for six different scenarios. Refer to Figure 10 for plots of the errors.

1. You have two classes, low (20) observations, two predictors which are uncorrelated, random, and normally distributed. Logistic, QDA, LDA, and Bayes are appropriate, but KNN isn't good since sample size is small.

2. Same as (1), except for the predictors each have a correlation that is the same. Then, logistic, LDA, and QDA but Bayes is not good anymore since there is correlation among predictors. KNN still not good.

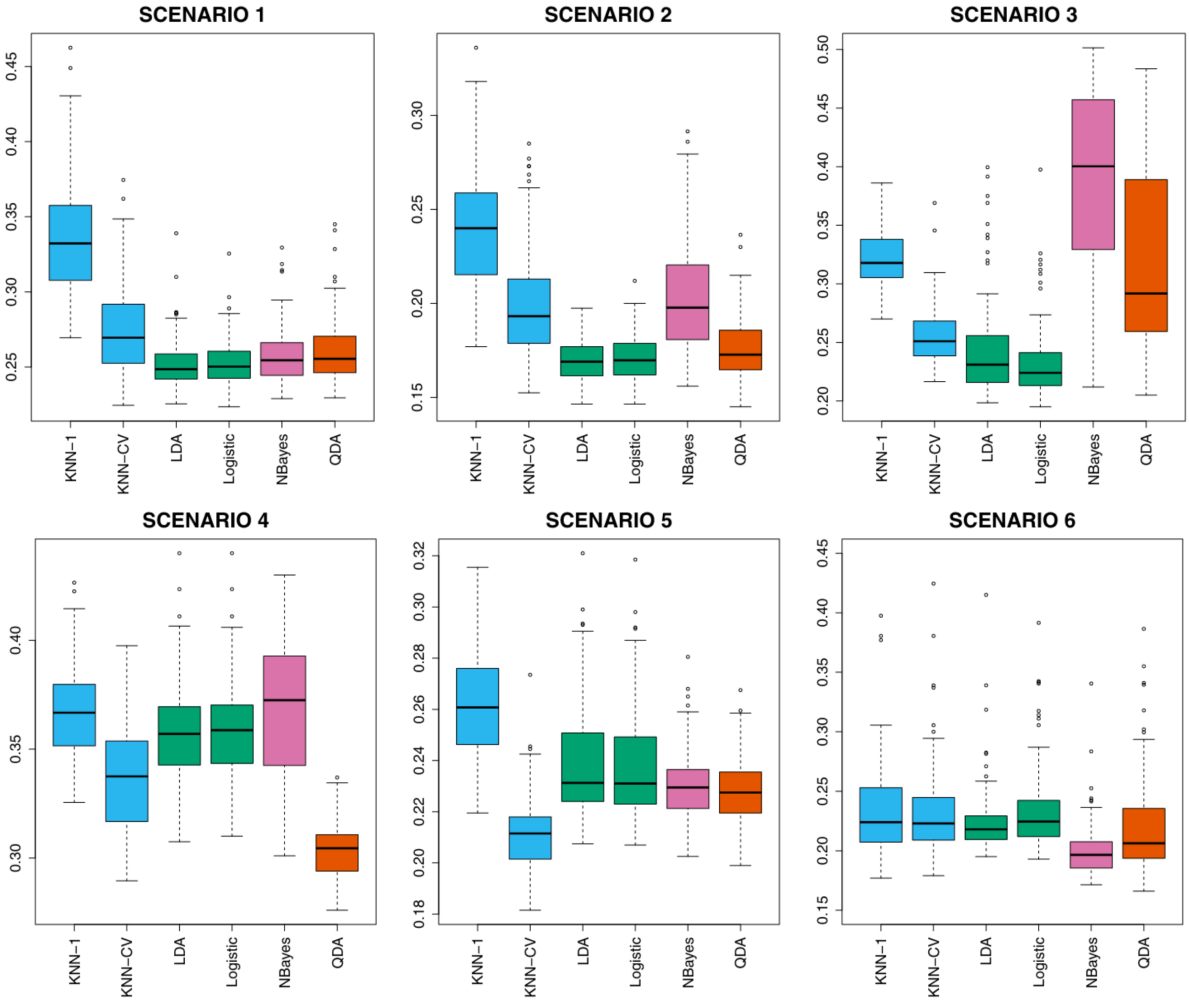
3. Same as (2), except for the predictors are drawn from the t-distribution, and there are 50 observations per class, which is different from a normal distribution. Then, everything does shit except for logistic.

4. Normally distributed, and each variable has its own correlation. Then, QDA cleans up.

5. Normally distributed predictors, and then classes were mapped to a really complicated non-linear function. Then, KNN outperforms everything with the right choice of K.

6. Very small sample size, each class has an unequal covariance matrix, variables uncorrelated and normally distributed. Then Bayes wins.

Figure 10: Comparison of classification methods with empirical examples.



Statistical Learning (Chapter 4) - Hannah's Notes

4.2 Why Not Linear Regression?
Coding of qualitative variables → numbers implies ordering of the outcomes. Ex possible diagnoses:

$$y = \begin{cases} 1 & \text{if stroke;} \\ 2 & \text{if drug overdose;} \\ 3 & \text{if epileptic seizure} \end{cases}$$

Reordering these would produce fundamentally different linear models → different sets of predictions. Also, when it comes to binary response / interpretation of results as probabilities, extrapolation in estimates may be outside interval → hard to interpret. Linear regression is even worse fit to predicting probabilities given more than 2 classes.

4.3 Logistic Regression
Consider the Default data set being used in this chapter (response falls into either Yes or No). Logistic regression models the probability that Y (response) belongs to a particular category.

Chapter 4 coding exercises

1. Use the credit dataset. Try to predict the ethnicity of the card user with the other independent variables. Assign each ethnicity to a category $(1, 2, 3)$.
   - First use logistic regression, and try a test-train split. Do this with single variable logistic regression and plot how well each one does
   - Do it rationally with no model selection algorithm (this time, multinomial regression obviously).
2. Try to implement forward selection with logistic regression.
3. Try to implement Monte Carlo with logistic regression, using forward selection as an initial guess.
4. Try the same computation except with new methods. Use the best two variables from logistic regression so you can plot them in 2D, and then move to higher dimensions later.
   - LDA
   - QDA
   - KNN

   Compute the error rates for each of the methods.

Solutions

Figure 11: The credit data was boring and it was hard to see differentiation among ethnicities, so I used high school test results. This is a raw plot of the school type based on classroom size and test score improvement.

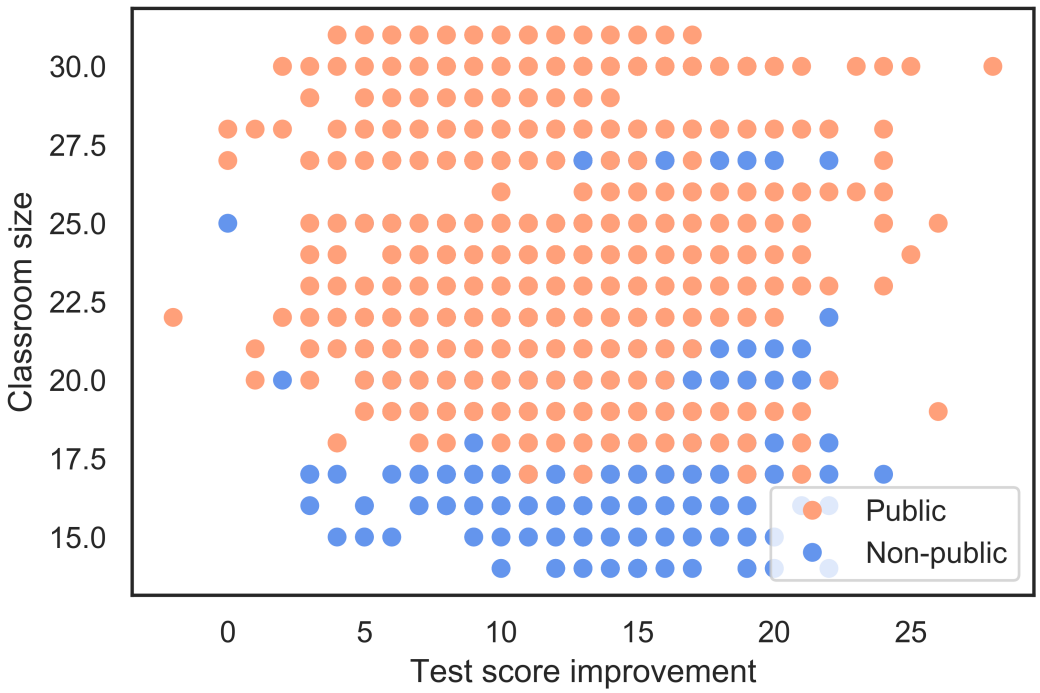


Figure 12: I performed logistic regression and made a decision boundary for this data. The error rate was not bad, only 17.8%. This was the logistic regression decision boundary for the school type dataset.

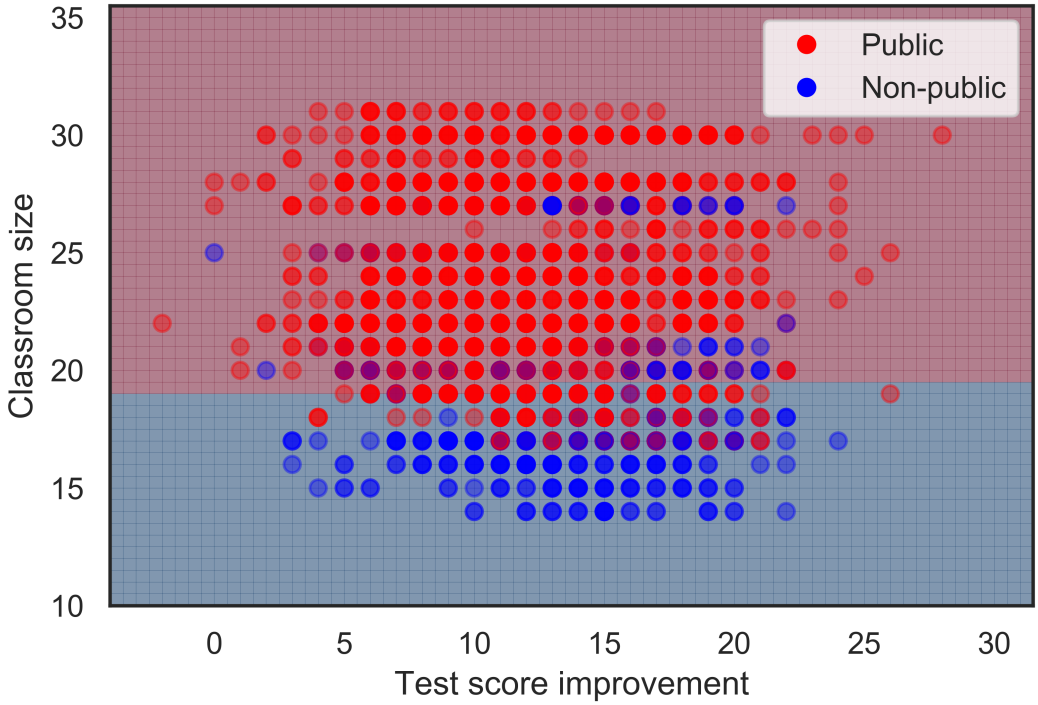


Figure 13: I wanted to spice things up with the logistic regression and multinomial regression, so I used a dataset from League of Legends 2020 Esports Match data. In League of Legends, apparently there are five classes your player can be: 'Fighter', 'Mage', 'Tank', 'Support', and 'Marksman'. To simplify things, first I looked at the two classes which seemed qualitatively most different: 'Tank' and 'Support'. I chose two variables rationally that made distinguishing easy on pairplots. Qualitatively, it seemed like attack damage and HP regeneration are good for clustering the two.

Figure 14: I performed logistic regression with a decent error rate, 9.8%.

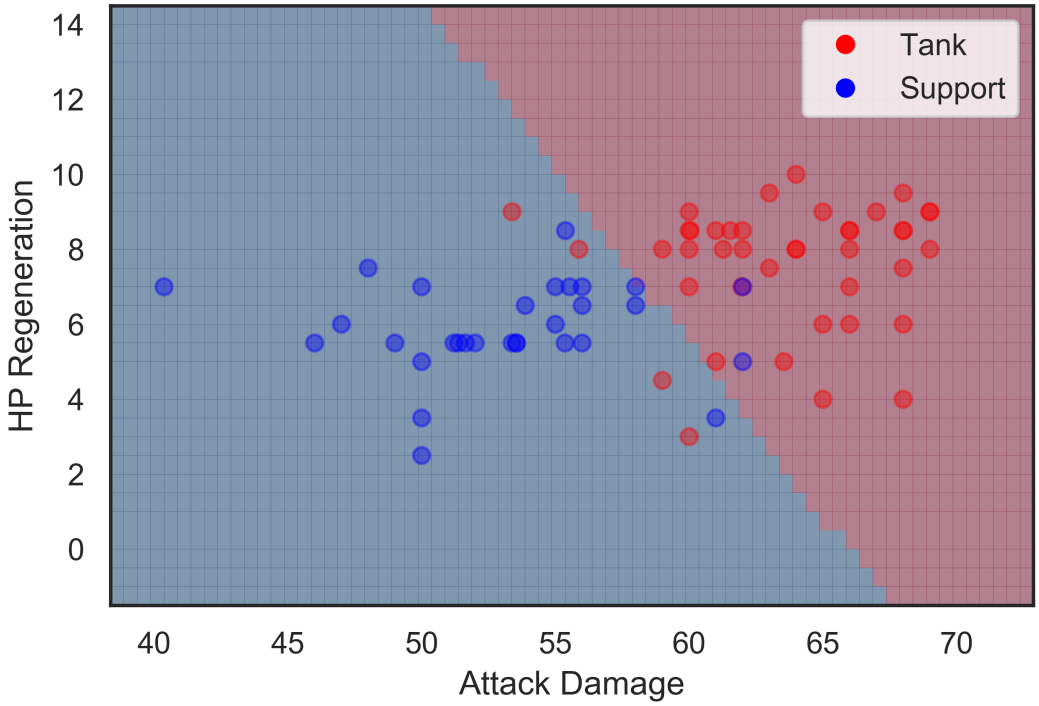


Figure 15: I was next curious how multinomial logistic regression would deal with things if there were multiple classes, like all five classes that are actually present in League. With all the classes and using the two variables that I already used, it seemed like finding decision boundaries would be pretty messy.

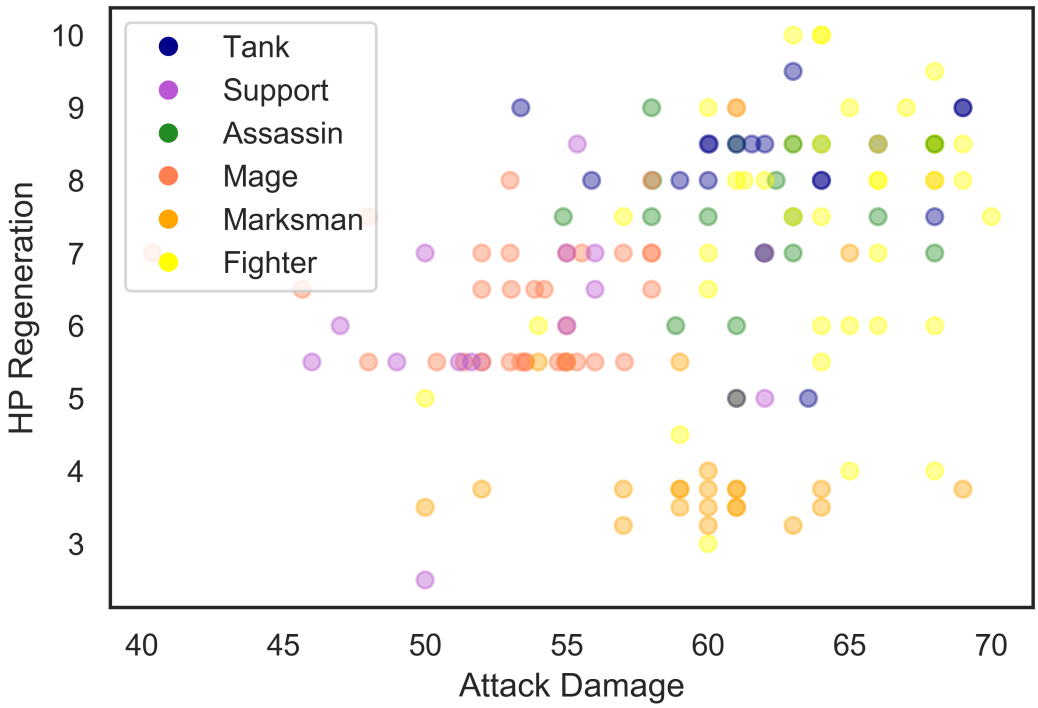


Figure 16: The multinomial regression tried to make some sensible classification boundaries, but had lots of trouble and even left out an entire class (Mage). The error rate was 45.3%.

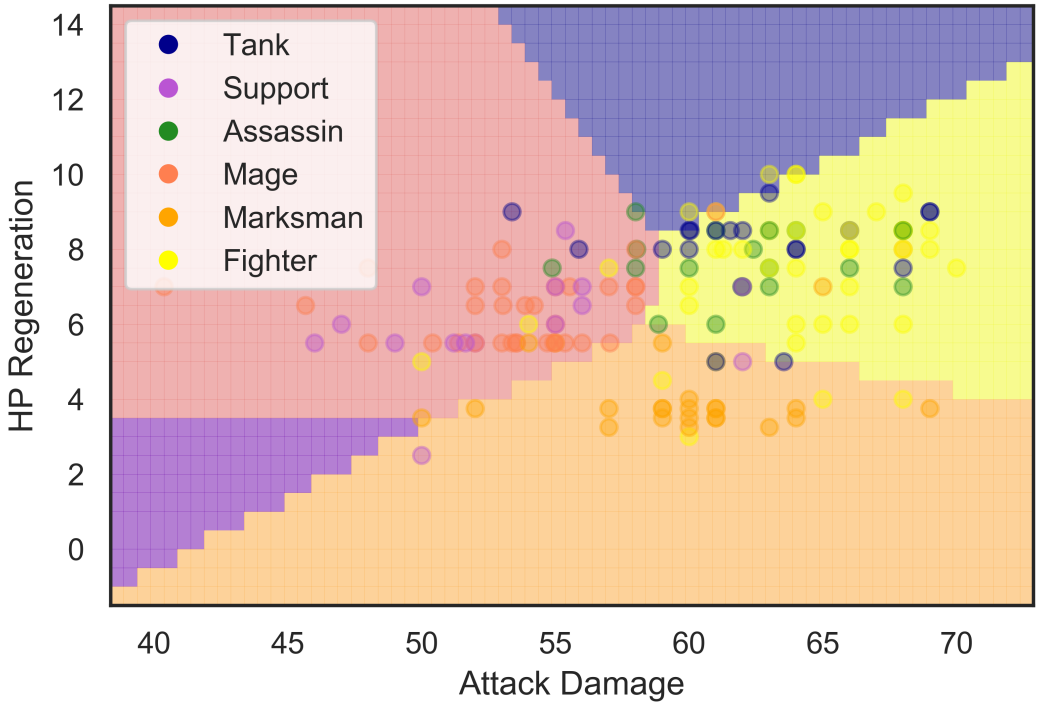


Figure 17: I went back to the pairplot with all of the possible classes to see if I could rationally choose a few variables that were best at classifying the character type. Just qualitatively, it seemed like attack damage, difficulty, hp, mp per level, and armor were the predictors that provided the best separation.

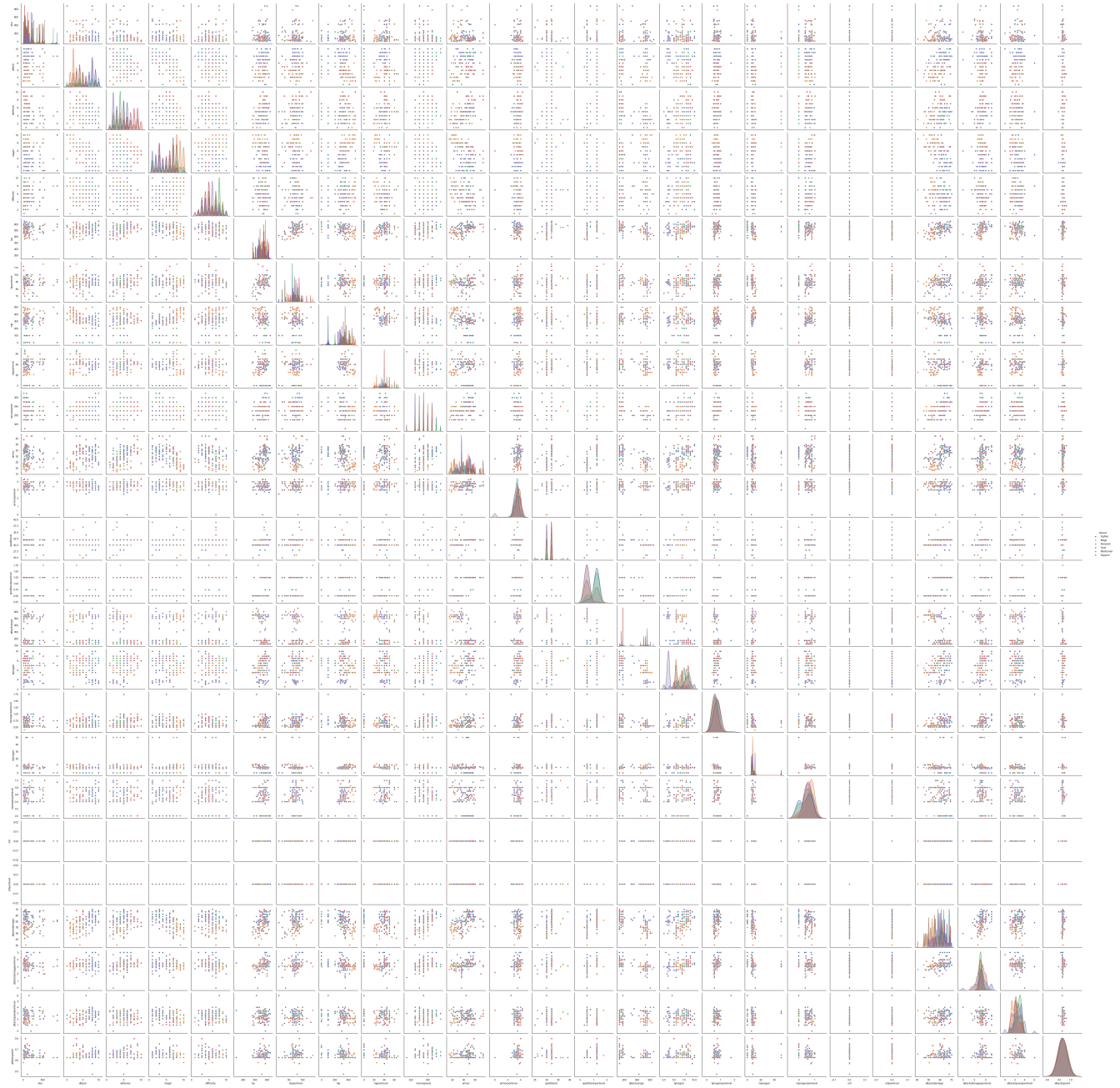


Figure 18: I performed a multinomial regression using the predictors ['attackdamage', 'difficulty', 'hp', 'mpperlevel', 'armor']. The error rate was much improved, but was still at 28.6%. I wondered then if I could use a greedy model generation.

```
"""
Qualitatively, it seems like the armor, mpperlevel, difficulty, hp, and attack
damage provide the best separation
"""
LOL['classes_int'] = classes_int

x=LOL[['attackdamage','difficulty','hp','mpperlevel','armor']]
y= LOL['classes_int']
log_reg = LogisticRegression(multi_class='multinomial', solver='lbfgs',  max_iter=10000)
model1= log_reg.fit(x,y)

ypred = model1.predict(x)
errors = []
for i in range(len(ypred)):
    if ypred[i] != np.array(LOL['classes_int'])[i]:
        errors.append(1)
    else:
        errors.append(0)

error_rate = np.sum(errors)/len(errors)
print("Error rate ="+str(error_rate))
```

Figure 19: I wrote a script to perform forward multinomial regression selection on the dataset, and ran it for a target variable number of 5 (to compete with my rational choice). The variables this found were ['armor', 'attack', 'attackrange', 'difficulty', 'defense']. The error rate was reduced to 20%.

```
def forward_selection(df, y, tar_predictors):
    column_names = np.array(df.columns)
    remaining_vars = df.shape[1]
    total_vars = df.shape[1]
    model_vars = 0
    chosen_var = []

    # run until you achieve the desire number of variables in model
    while model_vars <= tar_predictors:
        # initialize a list of RSS' for each of the variables tested
        error_list = []
        for i in range(remaining_vars):
            test_var = column_names[i]

            if remaining_vars == total_vars:
                # if starting from beginning, let X test just one variable
                X = df[[test_var]]
            else:
                # update vars from model
                X_list = []
                for j in range(len(chosen_var)):
                    X_list.append(chosen_var[j][0])
                X_list.append(test_var)
                X = df[X_list]

            regr = LogisticRegression(multi_class='multinomial', solver='lbfgs'
            regr.fit(X, y)
            #params = np.append(regr.intercept_,regr.coef_)
            predictions = regr.predict(X)

            errors = []
            for i in range(len(ypred)):
                if predictions[i] != y[i]:
                    errors.append(1)
                else:
                    errors.append(0)
            error_rate = np.sum(errors)/len(errors)
            error_list.append(error_rate)

        best_error = np.where(np.array(error_list) == min(error_list))
        chosen_var.append(column_names[[best_error[0][0]]])

        remaining_vars = remaining_vars-1
        column_names = np.delete(column_names, best_error[0][0])
        model_vars = model_vars + 1
```

Figure 20: I took it a step further and looked at how error rates were across the full range of variable choices using forward selection. The goal of this was to find the simplest model with the best error rate. I decided qualitatively that eight predictors was the best number, which were the following: [‘armor’, ‘attack’, ‘attackrange’, ‘difficulty’, ‘defense’, ‘mpperlevel’, ‘hpperlevel’, ‘attackdamage’]. If I were to summerize this, it would be that attack and defense attributes best separate the classes. This model only had a 10% error rate.

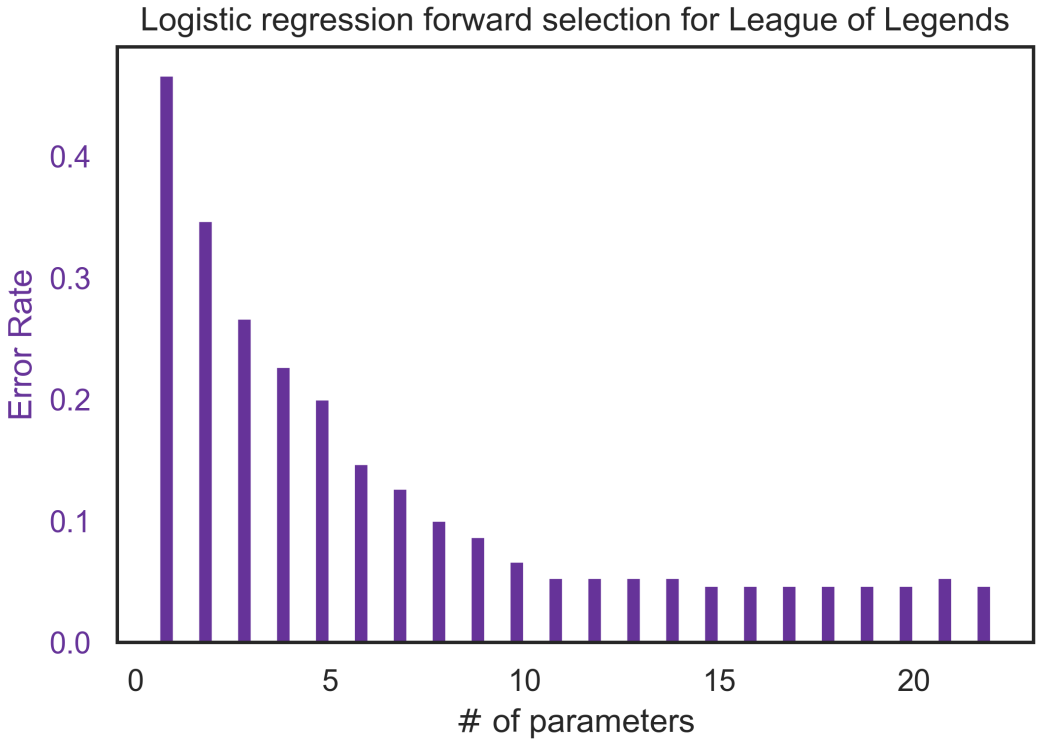


Figure 21: To test whether this selection method made results that were the best possible, I wrote a Monte Carlo script to sample other possible models. For four parameters, it was clear that the initial guess (forward selection) was the best one.

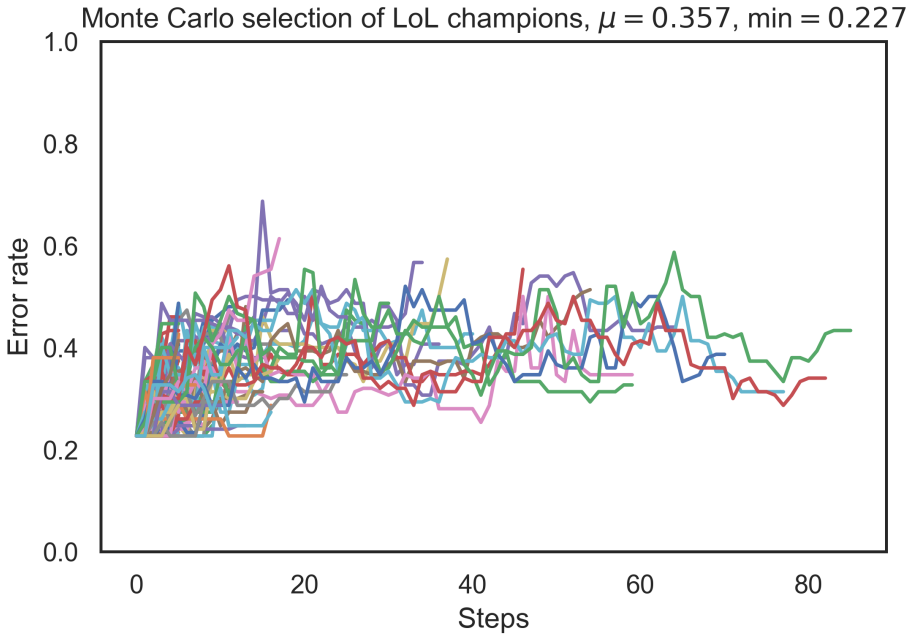


Figure 22: I tried with a model of 7 parameters, and the Monte Carlo was able to find a few models that had better error rates. It improve FS error rate from 12.7% to 10.7%. The predictors it used were [‘armor’, ‘attack’, ‘attackrange’, ‘defense’, ‘mpperlevel’, ‘hp’,‘attackdamage’]. It swapped essentially difficulty and hpperlevel for hp and attackdamage.

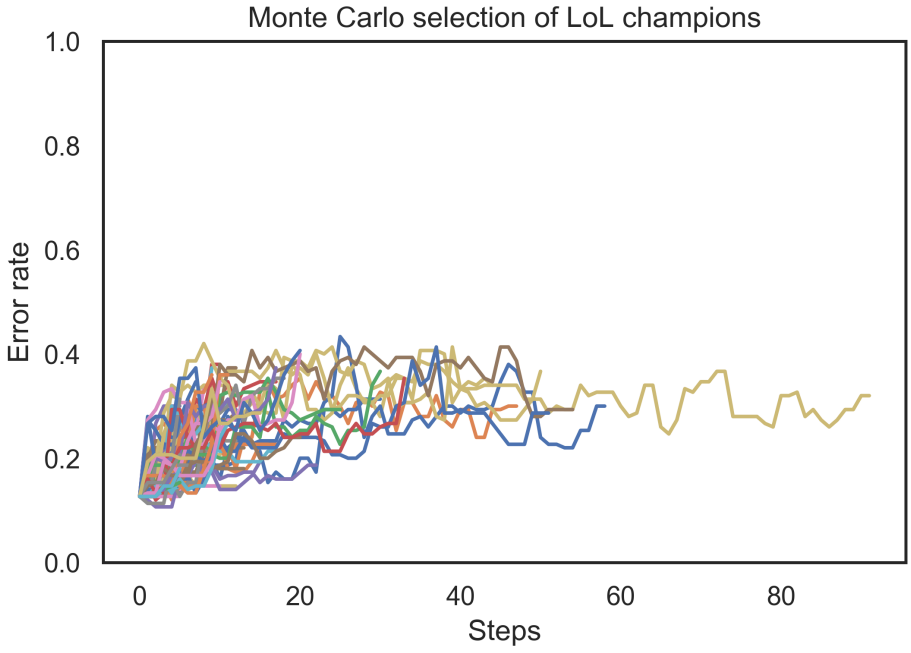
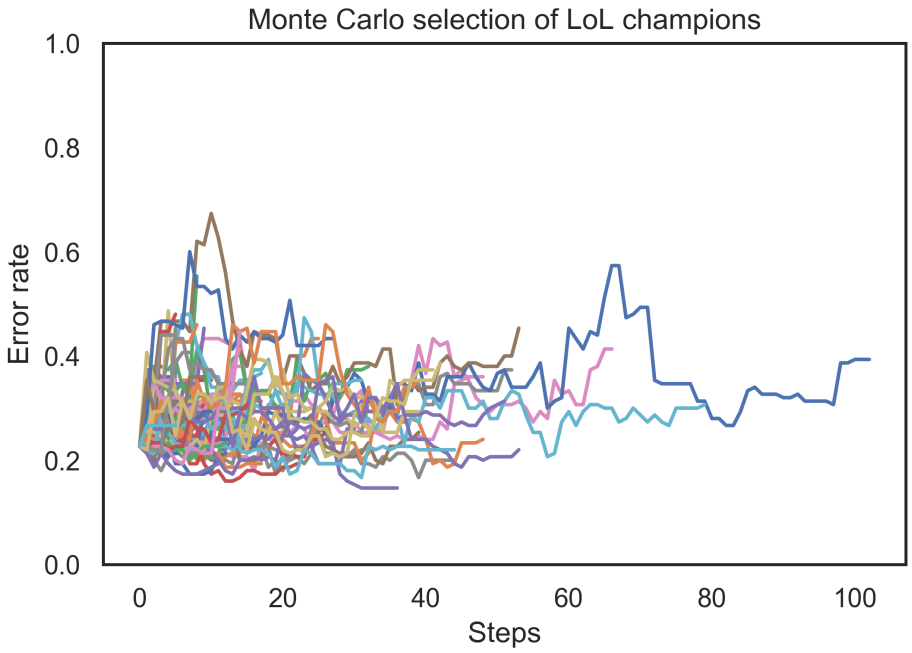


Figure 23: Finally I allowed the MC to add and delete variables randomly, and it was only really able to improve on the initial guess by adding tons of variables.



Chapter 10

Statistical Learning (Chapter 10) - Hannah's Notes

Deep Learning
Basics of neural networks and deep learning, and some of the specializations for problems such as CNNs for image classification and RNN for time series / other sequences.

10.1: Single Layer Neural Networks

Input: p variables in the form

$$X = (X_1, X_2, ...X_p)$$

Output: nonlinear function $f(X)$ to predict response Y. Distinguishing factor is that neural networks are composed of *input layers* which feed into each of K (chosen by the user) *hidden units*. The resulting activations of the K hidden units (expressed as $A_k, k = 1, ..., K$) are computed as functions of the input features (X), and are each essentially a different transformation $h_k(X)$ to a binary/logical scale of the original features.

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

where g(z) is a nonlinear *activation function* specified by the user. These feed into the output layer, and their sums result in a linear regression model with K activations:

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

All parameters ($\beta_0, ..., \beta_K$ and $w_0, ..., w_{Kp}$) need to be initialized with estimated values from the data. For a quantitative response, we can typically use squared-error loss (choosing parameters to minimize):

$$\sum_{i=1}^{n} (y_i - f(x_i))^2$$

See Section 10.7 for details about minimization.

For the activation function (g(z)), the traditional fn is the sigmoid (which converts a linear function into probabilities between 0 and 1):

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

Modern neural networks generally use the *ReLU (rectified linear unit)* activation function instead, which takes the form:

$$g(z) = (z)_+ = \begin{cases} 0 & z < 0 \\ z & \text{otherwise} \end{cases}$$

This can be computed and stored more efficiently than the traditional sigmoid activation. In our case, though the threshold is at 0, its application to the linear activation function $A_k \rightarrow$ shifted inflection point on account of the constant w_{k0}.

All this results in a model that derives K new features by computing K different linear combinations of X and then squishing them each through the activation function g(.) to transform them. It takes the form:

$$\begin{aligned} f(x) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j) \end{aligned}$$

Using a nonlinear activation function is important for keeping f(X) from collapsing into a simple linear model.

10.2: Multilayer Neural Networks

Example: the MNIST handwritten digits dataset. Every image is 28 x 28 (784) pixels, and each has a 0-255 value. These are stored in input vector X (in column order) and the output is the class label represented by vector Y of 10 dummy variables (1 in position corresponding to label, 0s elsewhere).
To solve this problem, apply a multilayer network that is different from the single layer in that:

> 2 hidden layers (L_1 = 256 units, L_2 = 128 units) [] 10 output variables, which represent one single qualitative variable. Generally, in *multitask learning* you can predict different responses simultaneously with a single network.
>
> Loss fn used for network training = tailored for multiclass classification.

First hidden layer is not too different from that in the single layer activation structure:

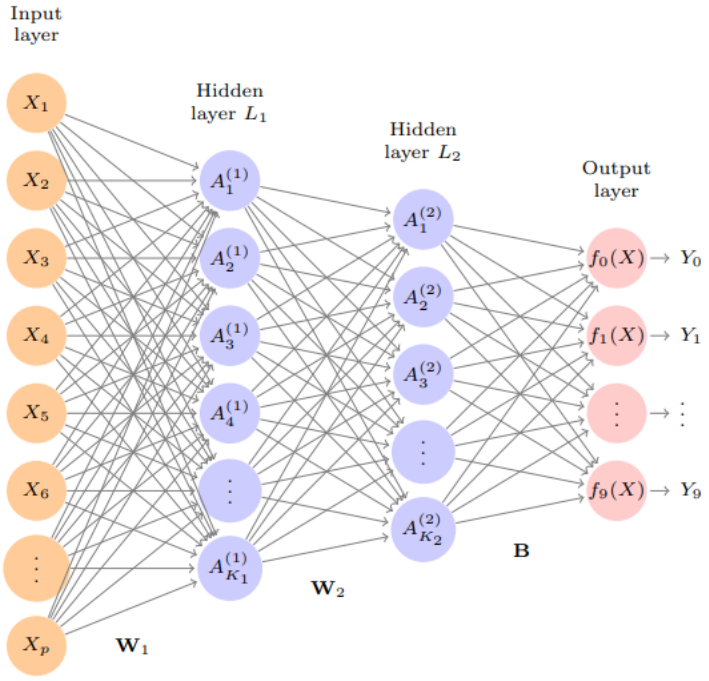
$$\begin{aligned} A_k^1 &= h_k^1(X) \\ &= g(w_{k0}^1 + \sum_{j=1}^{p} w_{kj}^1 X_j) \end{aligned}$$

Where superscript 1 is the indexing for the first layer, the neurons are k = 1, ..., K_1 (where K_1 would be the number of neurons, 256), and the sum from j=1 to p accounts for the weighted contribution of each X input (see Figure X). The second hidden layer treates the activations A_k^1 of the first hidden layer as inputs for computation of the activations in the second layer:

$$\begin{aligned} A_l^2 &= h_l^2(X) \\ &= g(w_{l0}^2 + \sum_{k=1}^{K_1} w_{lk}^2 A_k^1) \end{aligned}$$

where as with the first layer, superscript 2 is the indexing for the second layer, the neurons are l = 1,..., K_2 (where K_2 would be the number of neurons, 128), and the sum from k = 1 to K_1 accounts for the weight contribution of each of the K_1 previous neurons (see Figure X). However, the activations in this layer are still a function of X, though they are more complicated transformations.

Figure 24: Neural network diagram with two hidden layers and multiple outputs, for the MNIST handwritten digit problem. k in equations corresponds with the first hidden layer, l with the second. Ws represent the entire matrices of weights that feed from one layer to the next.



At the output layer, we now have 10 responses instead of 1 as we did in the single layer NN. Each is a linear model:

$$Z_m = \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} {h_l}^2(X)$$
$$= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} {A_l}^2$$

for m = 0, 1, ... 9 and the sum for l = 1 to K_2 accounts for each value from the second hidden layer that feeds into the output layer. The matrix **B** (see Figure X) store all 128 x 10 of these weights.
If all were separate quantitative responses, we would set each to $f_m(X) = Z_m$. However, we want our output to represent class probabilities $f_m(X) = \Pr(Y = m|X)$ (as in multinomial logistic regression. To transform our values into this, we use the softmax activation fn from 4.12, pg 141.

$$f_m(X) = \Pr(Y = m|X) = \frac{\exp Z_m}{\sum_{t=0}^{9} \exp Z_t}$$

for m = 0, 1, ...9. This function's transformations changes the results from the input and hidden layers into a probability for each of the 10 classes, and the classifier then selects the class with the highest probability. During training of this qualitative network, we want to find coefficient estimates that minimize the negative multinomial log-likelihood (cross-entropy):

$$-\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log f_m(x_i))$$

While the NN drastically outperforms multinomial logistic regression and linear discriminant analysis, it has 4 times as many coefficients observations in the training set. In order to avoid overfitting, can use two types of regularization: ridge regularization and *dropout* regularization.

10.3 Convolution Neural Networks

Consider the CIFAR100 database: 60k images labeled according to 20 superclasses (5 classes per superclass), each of resolution 32 x 32, with 3 RBG 8bit values for each pixel. First two axes are spatial (32 dimensional) and the third is the channel axis (RGB). CNNs first identifies low-level features such as small edges, patches of color, etc. These are combined to form higher-level features, which contribute to the probability of any given output class. In order to accomplish this, a CNN is composed of 2 specialized types of hidden layers: *convolution* (search for instances of small patterns) and *pooling* layers (downsample these instances to select prominent subset).

10.3.1 Convolution Layers

Convolution layer = large number of convolution filters (templates that determine whether local features are present in an image). These rely on convolutions (essentially repeatedly multiplying matrix elements and adding the results). Consider a simple example:

$$\text{original image } = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

Now consider a 2x2 filter of the form:

$$\text{convolution filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

Application of this filter to the image results in the matrix below (the filter is applied to every 2x2 submatrix in the original image in order to obtain the convolved image). If a 2x2 submatrix of the original image resembles the convolution filter, it will have a large value in the convolved image, and vv (e.x. *the convolved image highlights regions of the original image that resemble the convolution filter*).

$$\text{convolved image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

With CNN, the filters are *learned* for the specific classification task: the weights are essentially the parameters going from input $\rightarrow$ hidden layer, with one hidden unit per pixel in the convolved image. More details for this example:

> The RGB channels are represented by a 3D array. Each channel is a 2D (32x32) feature map, and the corresponding filter will also have 3 channels (one per color), each 3x3 with potentially different filter weights. Results of the 3 convolutions = summed to form a 2D output feature map.

Using K different convolution filters at the 1st hidden layer → K 2D output feature maps → single 3D feature map. Textbook says to consider this just as the activations in a hidden layer of a simple neural network, but organized / produced in a spatially structured way.

Typically apply ReLU activaiton fn to convolved image (sometimes referred to as a *detector layer*).

10.3.2 Pooling Layers

Provides a way to condense a large image into a smaller one. *Max pooling* operation (for example) summarizes each non-overlapping 2x2 block of pixels in an image using the max value in the block. E.x.:

$$\text{max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

10.3.3 Architecture of a CNN

Input layer = 32x32 2D map that represents color (channel axis) → after 1st round of convolutions, feature map with the number of channels = number of convolution filters used. This is followed by max-pool layer (reduces size of feature map). Repeated for next two layers:

Each subsequent convolve layer also takes input (3D map from previous layer) and treats it as single multi-channel img.

Since channel feature maps = smaller after each pool layer, generally increase the number of filters in next convolve layer to compensate.

Can repeat several convolve layers before pool layer to effectively increase the dimension of the filter.

These are repeated until pooling → each channel feature map to just a few pixels in each dimension, which are then fed into 1+ fully connected layers before the output layer (flattened into one group).

10.3.4 Data Augmentation

Essentially replicating training images with random distortion so that the training set is increased and protected against overfitting.

10.4 Document Classification

How to featurize (classify with adjectives based on content) different documents? Most basic = *bag of words* model.

Score each document of presence or absence of words in a language dictionary (for M words in the dictionarry, we create a binary list of M where 1 = present, and 0 otherwise).

To ensure that the list is manageable, limit the dictionary to, for ex., the 10k most common words in 25k reviews that need to be classified.

For a given reivew, the binary list will be mostly 0s and some 1s in positions that correspond to words present in the document. This is *sparse matrix*, since most of the values are the same and it can be stored efficiently.

Passing this sort of processed data through a 2-class NN with 2 hidden layers each with 16 ReLU units results in similar fitting as with using a lasso logistic regression (a 2-class NN amounts to a nonlinear logistic regression model). Both have a tendency to overfit.

$$\log \frac{Pr(Y=1|X)}{Pr(Y=0|X)} = Z_1 - Z_0 \tag{1}$$

$$= (\beta_{10} - \beta_{00}) + \sum_{l=1}^{K_2} (\beta_{1l} - \beta_{0l}) {A_l}^2 \tag{2}$$

This model summarizes documents by noting the presence of words and ignores their context. Some ways to take the context into account:

bag-n-grams model records consecutive co-occurences of every distinct n member group of words.

Treat the document as a seequence (take account all of the words in the context of preceding and following).

10.5 Recurrent Neural Networks

Sequential data requires special treatment when it comes to building models for prediction. Examples include:

Documents: relative positions of words are clues as to the narrative, theme, etc. We can exploit this when it comes to topic classification, sentiment nalysis, and language translation.

Time series for weather forecasting

Financial times series data

Recorded speech, music, etc. Useful when it comes to transcription, translation, assessment, and attribute assignment

Handwriting, when it comes to turning it into digital text or reading (optical character recognition)

In an RNN, sequence X ($X = X_1, X_2, ..., X_L$) is the input object. Output Y can also be a sequence, but is often a scalar (consider binary sentiment label for documents) (see Figure 25).

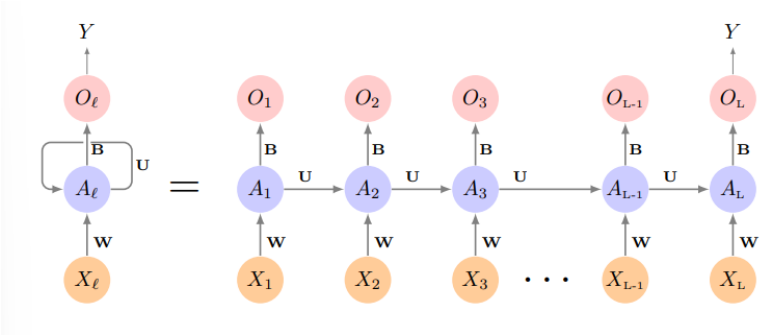
Suppose each vector X_l of the input sequence has p components such that $X_t^l = (X_{l1}, X_{l2}, .., X_{lp})$ and the hidden layer consists of K units $A_t^l = (A_{l1}, A_{l2}, .., A_{lK})$. We represent the collection of K X (p+1) shared weights w_{kj} for the input layer by a matrix W. Similarly, U is a KxK matrix of the weights u_{ks} for hidden - hidden layers, and B is a K + 1 vector of weights β_k for the output layer such that the activation is the base weight plus the sums of the outputs from W and U:

$$A_{lk} = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{lj} + \sum_{s=1}^{K} u_{ks} A_{l-1,s})$$

and the output O_l is computed as:

$$O_l = \beta_0 + \sum_{k=1}^{K} \beta_k A_{lk}$$

Figure 25: Schematic of a simple RNN. Input is vector sequence (X_{l1}^{L}) and Y is a single response. This RNN processes X sequentially: each X_l feeds into the hidden layer to produce the current activation vector A_l, which has as input the activation A_{l-1} from the previous element in the sequence. The same group of weights (W, U, B) are used as each element of the sequence is processed, and each layer produces a sequence of predictions O_l from the current activation A_l but generally we only care about the last one, which results in the output Y.



for a quantitative response (we add an additional sigmoid activation function for a binary response), where g(.) is an activation function such as ReLU. Note that W, U, B are used for processing each element in the sequence (they are NOT functions of l): this *weight sharing* is similar to filter use in CNNs (accumulate history so that context can be used for prediction).

In regression problems, loss fn is:

$$(Y - O_l)^2$$

which only references final output. For RNN, each element X_l contributes to O_l and thus indirectly to W, U, and B. With n input sequence/response pairs (x_i, y_i), we can find the parameters by minimizing the sum of squares:

$$\sum_{i=1}^{n}(y_i - o_{iL})^2 = \sum_{i=1}^{n}(y_i - (\beta_0 + \sum_{k=1}^{K}\beta_k g(w_{k0} + \sum_{s=1}^{K} u_{ks} a_{i,K-1,s})))^2$$

where y and x here correspond to output Y and input X. The intermediate outputs provide an evolving prediction for the output, and is needed for some learning tasks.

10.5.1 Sequential Models for Document Classification

When using the bag-of-words model, we can represent each word in an *embedding* space in order to reduce the data's dimensionality. Rather than representing each word with a binary vector, we can represent it by a set of m real numbers so that we have a new matrix E that is m x 10k, where each col is indexed by one of 10k words in a dictionary and the values in the column give m coordinates for that word in the embedding space. oh my

If we have a large dataset, we can have the NN learn E (the embedding layer) or insert a precomputed E into the embedding layer that was calculated through PCA. In the textbook example, pretrained embeddings are used that result in positions of words that preserve semantic meaning. Each document is represented as a sequence of m-vectors that represents the sequence of the last L words (L vectors X = $X_1, X_2, ..., X_L$ where each has m components). More details in txtbook

skipped 10.5.2 Time Series Forecasting

10.5.3 Summary of RNNs

Lots of variations:

> Can use 1D CNN and treat the vector sequence as an image. CNN slides along sequence and can learn phrases / subsequences relevant to learning task
>
> Can have additional hidden layers, as well as bidirectional RNNs that scan in both directions

10.6 When to Use Deep Learning

When faced with several methods that give roughly equivalent performance, pick the simplest. Expect deep learning to be most helpful when the sample size (training set) is extremely large, and when interpretability of the model is not high priority.

10.7 Fitting a Neural Network

Take the simple NN we talked about in section 10.1. The parameters are $\beta = (\beta_0, \beta_1, ..., \beta_K)$ and $w_k = (w_{k0}, w_{k1}, ..., w_{kp})$ and $k = 1, .., K$. With observations $(x_i, y_i), i = 1, ..., n$, fitting the model can be done through nonlinear least squares:

$$\underset{w_k{}_1^K, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij})$$

Nested arrangement + symmetry = nonconvex problem with multiple solutions. To combat issues + protext from overfitting, we use:

> *Slow Learning*: relatively slow iterative fashion with gradient descent, where the fitting process is stopped once overfitting is detected.
>
> *Regularization*: penalties are imposed on parameters (lasso or ridge, 6.2)

Let all the parameters be in one long vector ϑ, so that the minimization can be rewritten as:

$$R(\vartheta) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f_\vartheta(x_i))^2$$

where f explicitly depends on the parameters. We can then do gradient descent to search for the local minimum:

1. Begin with guess ϑ^0 for all parameters in ϑ, and let t = 0

2. Iterate until $R(\vartheta)$ (the objective) does not decrease — decreases less than tolerated:

   Find a vector δ that reflects a small change in ϑ, such that:

   $$\vartheta^{t+1} = \vartheta^t + \delta$$

   reduces $R(\vartheta)$, and set t $\leftarrow$ t + 1

10.7.1 Backpropagation

In order to find the directions in which to adjust ϑ to decrease the objective, we can find the gradient of $R(\theta)$ evaluated at some current value ϑ^m:

$$\nabla R(\vartheta^m) = \frac{\partial R(\vartheta)}{\partial \vartheta}\Big|_{\vartheta=\vartheta^m}$$

This gives the direction in ϑ-space where the objective increases the most rapidly, and we go a little ways ϱ (the learning rate) in the other direction. If the value is 0, then it is possible that we arrived at a minimum.

Calculating the gradient = chain rule! As $R(\vartheta) = \sum_{i=1}^{n} = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\vartheta(x_i))^2$ is a sum, its gradient is also a sum over n observations. Take one of the terms as an example:

$$R_i(\vartheta) = \frac{1}{2}(y_i - \beta_0 - \sum_{k=1}^{K}\beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}))^2$$

Where we can express $\sum_{j=1}^{p} w_{kj}x_{ij}$ as w_{ik}. First, take the derivative with respect to the parameter (β_k):

$$\frac{\partial R_i(\vartheta)}{\partial \beta_k} = \frac{\partial R_i(\vartheta)}{\partial f_\vartheta(x_i)} \cdot \frac{\partial f_\vartheta(x_i)}{\partial \beta_k} \tag{3}$$

$$= -(y_i - f_\vartheta(x_i)) \cdot g(z_{ik}) \tag{4}$$

Then, take the derivative with respect to the weight (w_{kj}):

$$\frac{\partial R_i(\vartheta)}{\partial w_{kj}} = \frac{\partial R_i(\vartheta)}{\partial f_\vartheta(x_i)} \cdot \frac{\partial f_\vartheta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} \tag{5}$$

$$= -(y_i - f_\vartheta(x_i)) \cdot \beta_k \cdot g(z_{ik}) \cdot x_{ij} \tag{6}$$

Note that both of the partials have the residual $y_i - f_\vartheta(x_i)$, but the one with respect to the parameter has a fraction of the residual attributed to each of the hidden units according to the value of $g(z_{ik})$, while the one with respect to the weight has a similar attribution to input j via hidden unit k. We can see that the gradient assigns a fraction of the residual to each of the parameter classes (*backpropagation*).

10.7.2 Regularization and Stochastic Gradient Descent

When n is large, we can sample a small fraction or *minibatch* of them every time we compute a gradient step (stochastic gradient descent). Textbook says to use premade software. Consider the multilayer network we used in the MINIST problem, which has around 235k weights (need regularization to avoid overfitting!). We can do this by augmenting the objective function with a penalty term:

$$R(\vartheta : \lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im} \log f_m(x_i)) + \lambda \sum_j \vartheta_j^2$$

λ is often preset with a small value or calculated with the validation-set approach (5.3.1). We can use different values of λ for weight groups from different layers. Early stopping once we see that the validation objective begins to increase can also be used as a form of regularization.

10.7.3 Dropout Learning

New form of regularization that is similar to random forests (8.2) in that you remove a fraction Φ of the units in a layer when fitting the model. This is done separately each time a training observation is processed, and the suriviing ones compensate + their weights are scaled up by a factor of $\frac{1}{1-\Phi}$. This prevents nodes from over-specialization. In practice, we can randomly set the activations for the 'dropped out' units to zero, but make sure to keep the architecture intact.

10.7.4 Network Tuning

Though theorectically the networks we have been working with are straightforward, these choices all have an effect on the performance:

Number of hidden layers and units per layer: "Modern thinking" is that we can have lots of units in hidden layers and control for overfitting through regularization.

Regularization tuning parameters: include the dropout rate Φ and the strength λ of lasso and ridge regularization. Typically set independently at each layer.

Details of stochastic gradient descent: batch size, number of epochs, and (op) details of data augmentation.

10.8 Interpolation and Double Descent

One implication of the bias-variance trade off is that it's usually not a good idea to *interpolate* (get no training error) as that will likely result in a high test error. However, in specific situations you can do this .... double descent (error increases and then decreases again). Consider example where we simulate n = 20 observations from the model

$$Y = sin(X) + \varepsilon$$

Where X U[-5, 5] (uniform distribution) and ε $N(0, \varsigma^2)$ where $\varsigma = 0.3$. Fit natural spline (or least squares regression of the response onto a set of d basis functions) with d dof... see txtbk for rest of the explanation.

youtube ML: Backpropagation Calculus

Introduction

Consider a simple network with 4 layers, where each layer has one neuron in it. Network is determined by 3 weights and 3 biases:

$$C(w_1, b_1, w_2, b_2, w_3, b_3)$$

How sensitive is the cost function to these variables?

1 dimensional example

Consider the connection between the last two neurons. Let activation of the last one be marked with index L (layer indication), so activation of the previous one is a^{L-1}, and the determining weight is w_{L-1}. Let the desired output be y, so that cost (C_0) of this connection is:

$$C_0(...) = (a^L - y)^2$$

And define a^L to be:

$$a^L = \varsigma(w^L(a^{L-1}) + b^L)$$

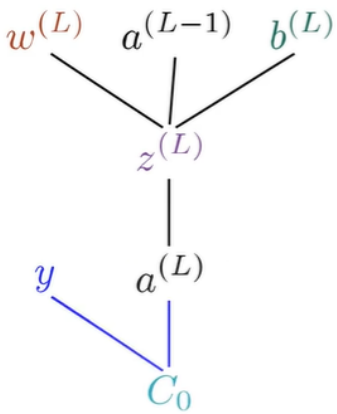
where ς is some wraparound function that expresses nonlinearity. To make our lives easier, let

$$z^L = w^L(a^{L-1}) + b^L$$

so that:

$$a^L = \varsigma(z^L)$$

Figure 26: Picture of how the functions feed into each other



Defining the relations of our functions with partials

Our question: what is the derivative of the cost function with respect to w^L ($\frac{\partial C_0}{\partial w^L}$)? I.e. how much does change in the w function influence the corresponding cost function (what is the ratio of the two)?
we can define this derivative as a chain (uh oh) of partials describing the hierarchical relations between the functions that we are using to define / inform our cost function so that:

$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C0}{\partial a^L}$$

If you refer to Figure 10, you can see that each fraction contains the partial of layer L over that of layer $L-1$. This is the chain rule. Sad.

Return to the example: last two nodes in the network

Now we can, using our formula, find the derivative of the cost function between the last two nodes. Recall that:

$$C_0 = (a^L - y)^2$$

Taking the derivative with respect to a^L gives us the expression for the third term for $\frac{\partial C_0}{\partial w^L}$:

$$\frac{\partial C0}{\partial a^L} = 2(a^L - y)$$

and tells us that a^L has a fairly large effect on our final product. Similarly, we can find that:

$$\frac{\partial a^L}{\partial z^L} = \varsigma'(z^L)$$

from our earlier definition of a^L to be $\varsigma(z^L)$. Likewise,

$$\frac{\partial z^L}{\partial w^L} = a^{L-1}$$

can be obtained from:

$$z^L = w^L(a^{L-1} + b^L$$

Therefore, the derivative of the cost function with respect to w^L (expression of the dependence of the latter on changes in the former) is:

$$\frac{\partial C_0}{\partial w^L} = a^{L-1}\varsigma'(z^L)2(a^L = y)$$

Expansion to complete cost function And the derivative of the *complete* cost function for the entire network can be written as the average of all the training examples:

$$\frac{\partial C}{\partial w^L} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial C_k}{\partial w^L}$$

This itself is only one component of the gradient, which is built of the derivatives of the cost function with respect to the different weights and biases for each layer:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^1} \\ \frac{\partial C}{\partial b^1} \\ ... \\ \frac{\partial C}{\partial w^L} \\ \frac{\partial C}{\partial b^L} \end{bmatrix}$$

We can obtain the expressions for these partials with a simple change to our previus definition of the partial of the cost function with respect to w^L. The derivative of cost with respect to the bias is simply:

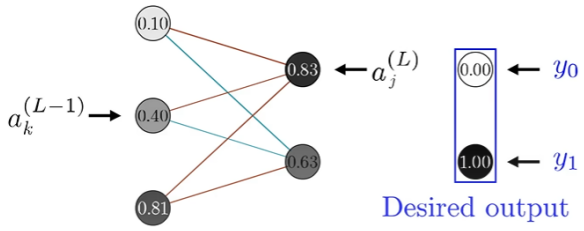
$$\frac{\partial C}{\partial b^L} = \frac{\partial z^L}{\partial b^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C_0}{\partial a^L} = 1\varsigma'(z^L)2(a^L = y)$$

You can see how sensitive the cost function is to the activation of the previous layer. Namely, if you then take the partial of the C_0 with respect to the previous layer (a^{L-1}), it can be seen that the partial of z^L with respect to the previous layer is simply the weight (w^L).
We can recurse through the layers to determine the relationships / effects of each layer and their associated weights and biases.

2 dimensional example

With more nodes in each layer and subsequently more outputs, we have to add a subscript as an additional index to keep track of which node we're at within a layer (Figure 11).

Figure 27: Schematic showing 2D network and relevant indexing



where the edge connecting ${a_k}^{L-1}$ to ${a_j}^L$ has a weight that is indexed as ${w_j k}^L$. Just as before, we can call that term simply ${z_j}^L$, where it is:

$${z_j}^L = {w_j 0}^L {a_0}^{L_1} + ... + {w_j k}^L {a_k}^{L-1} + {b_j}^L$$

and:

$${a_j}^L = \varsigma({z_j}^L)$$

and similarly for the other equations. The chain rule for the dependence of the cost function on ${w_j k}^L$ also does not look too different from the one derived above:

$$\frac{\partial C_0}{\partial {a_k}^{L-1}} = \frac{\partial {z_j}^L}{\partial {w_j k}^L}\frac{\partial {a_j}^L}{\partial {z_j}^L}\frac{\partial C_0}{\partial {a_j}^L}$$

However, things are different when it comes to the dependence of the cost function on the previous layer, since it is influenced by multiple paths. To account for this, we have to sum up the partials over the layer:

$$\frac{\partial C_0}{\partial {a_k}^{L-1}} = \sum_{j=0}^{n_L-1} \frac{\partial {z_j}^L}{\partial w_j {k}^{L-1}} \frac{\partial {a_j}^L}{\partial {z_j}^L} \frac{\partial C_0}{\partial {a_j}^L}$$

These partials determine each component of the gradient ∇C.

For the cost function itself, we will continue to use the distance from y_0 and y_1 to calculate. Recall that with multiple, we find the sum of the squared difference between the last layer activation and the desired output:

$$C_0 = \sum_{j=0}^{n_L-1} ({a_j}^L - y_j)^2$$

youtube: Intro to Object Detection in Deep Learning

Part 1