Chapter 9

Linear Regression

9.1 Introduction

In this class, we have looked at a variety of different models and learning methods, such as finite state machines, sequence models, and classification methods. However, in all of the cases that we have looked at so far, the unobserved variables we have been trying to predict have been finite and discrete. When we looked at Naive Bayes, we tried to predict if something was in a positive or a negative class. When we looked at Hidden Markov Models and Conditional Random Fields, we tried to figure out the part-of-speech tags of individual words. These methods have been attempting to predict which class a particular latent variable belongs to. For Naive Bayes, we motivated out problem with a decision between two classes. Though that number is arbitrary and you could easily add a third class called neutral, or have a problem that naturally has dozens of classes. For our part-of-speech sequence tagging in English, we often choose from a set of 36 tags. The common thread is that in all of these methods, we are trying to choose a set number of classes.

We now move on to look at what happens when we care about predicting a value that is not from a discrete set of possibilities, but rather, is continuous. For instance, if we are given an essay written in English, rather than predicting a letter grade, could we instead predict the percent score? Or, given a movie review, could we predict the average number of stars users rate it, rather than just saying if the review is positive or negative?

9.2 Linear Regression

Linear regression is a powerful tool used for predicting continuous variable outputs. In general, to predict an output that is continuous, the goal is to find a function that transforms an input into an output. Linear regression is simply the method that finds a solution to this problem by finding a linear function.

Generally, given an input x, we are trying to find a function, $f(x)$, that predicts an output y. This is the same way to formulate many of the learning methods that we have discussed in this class. We would like to find $f(x) = y$ for some observed data x that predicts unknown data y. In methods discussed previously, like Naive Bayes, y is often a class, such as 0 or 1, and cannot take any other values. For linear regression, we still would like to find a function, but now y can take on a range of values.

9.2.1 Supervised Learning

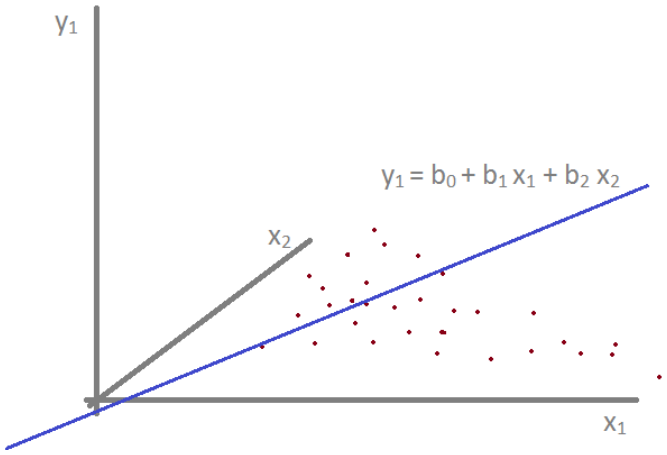
The majority of the models that we have discussed in this class are supervised, and linear regression is no different. Supervised learning models are simply methods that are trained using a data set where the values we are trying to predict (y) are known. After training on data where the values are known, the model tries to predict values for data that is unknown. In the classification tasks we have looked at (Naive Bayes, Logistic Regression, the Perceptron, and Topic Modeling) all of them are Supervised Learning methods except for Topic Modeling. In your homework, you have always been given supervised learning problems where the data contains a training file. You then report your predicted output compared to the the gold-standard labels in your testing file. Linear Regression is no different. Again, it is a method for predicting unseen values on a test set, having been given known values in a training set. The difference is that the output values are continuous are our learned function is linear.

9.3 Linear Models

Formally, we define Linear Regression as predicting a value $\hat{y} \in \mathbb{R}$ from m different features $\mathbf{x} \in \mathbb{R}^m$. We would like to learn a function, $f(\mathbf{x}) = y$ that is in a linear form. Specifically, we are interested in finding:

$$\hat{y} = \beta_0 + \mathbf{x}^\intercal \boldsymbol{\beta}$$

Given training n training examples $\langle \mathbf{x_i}, y_i \rangle$ for $1 \leq i \leq n$, where each $\mathbf{x}$ has m features, our goal is to estimate the parameters $\langle \beta_0, \boldsymbol{\beta} \rangle$.



9.3.1 Parameter Estimation

The goal in linear regression is to find a line that generalizes our data well. In other words, we would like to find $\langle \beta_0, \boldsymbol{\beta} \rangle$ that minimize the error in our training set. We do this by minimizing the sum of the squared errors:

$$\hat{\boldsymbol{\theta}} = \underset{\theta = \langle \beta_0, \boldsymbol{\beta} \rangle}{\arg\min} \frac{1}{2n} \sum_{i=1}^{n} (y_i - (\beta_0 + \mathbf{x_i}^\intercal \boldsymbol{\beta}))^2$$

The intuition here is that we would like to minimize the difference between y_i and the value predicted by $\beta_0 + \mathbf{x_i}^\intercal \boldsymbol{\beta}$. In other words, $\mathbf{x_i}$ is linearly transformed by $\boldsymbol{\beta}$ and β_0 to give a predicted value for y_i. Across all n training examples, we want the sum of the difference between our predicted and actual values to be at a minimum.

We need to estimate the values of our $\boldsymbol{\beta}$. To do this, we take the cost function we just defined and apply the gradient descent algorithm to our problem. This requires taking a partial derivative with respect to each of our m number of β's. After some algebraic manipulation we are left with the LMS or Least Mean Squares update rule. For more information on gradient descent and deriving the update rules, the text book "Pattern Recognition and Machine Learning" has some nice explanations beyond the scope of this lecture (Bishop 2009).

9.3.2 Features

In our definition, we said that we would like to find y given m features. We must define a set number of features and the assumption in linear regression is that they are independent. Linear regression then finds a linear combination of these features that best predicts our training data. We have defined our general function to be:

$$\hat{y} = \beta_0 + \mathbf{x}^\intercal \boldsymbol{\beta}$$

For the case where $m = 1$, or in other words, we only have one feature, our function is merely:

$$\hat{y} = \beta_0 + x_1\beta_1$$

This is simply the function for a line $y = mx + b$ where $m = \beta_1$ and $b = \beta_0$. Simply put we are mapping a single dimensional input to another dimension (x to y). We defined $x \in \mathbb{R}^m$ and for the case where $m = 1$, we are simply defining x to take a real value.

For the slightly more difficult case where $m = 2$, we are now mapping values from a 2-dimensional space into another dimension. In other words, with 2 features for each data point, our function reduces to:

$$\hat{y} = \beta_0 + x_1\beta_1 + x_2\beta_2$$

This is also a line (hence the term linear regression). However, this is now in 3-dimensional space. In general, we are mapping from an m-dimensional space to a single dimensional, continuous value space.

NLP Features

In some form or another, many of you will be familiar with linear regression, even if that is from a high school science class where you fit a line to some data points in an experiment. However, most of you will be familiar with this problem from an application different from Natural Language Processing, particularly one where your $\mathbf{x}$ values are real numbers, in a continuous space themselves. To make use of linear regression in NLP, we must also make sure that our values are real numbers. That is why we defined $\mathbf{x} \in \mathbb{R}^m$.

Mapping our features to real values is actually not a difficult problem, and many features we are interested in are already real numbers. For instance, getting computers to automatically grade essays is a topic of increasing interest, especially for standardized tests. We may hypothesize

(correctly) that the length of the essay is a good predictor for the score it receives. Counting the number of words in an essay gives us a real number that we can use as a feature. Additionally, we may consider average word length to be an adequate proxy for vocabulary size - or even just use the number of unique words in the essay. Again, these are already real valued variables.

However, we may also care about other features that are not inherently numbers, for instance part-of-speech tags or certain words. In this case, we can simply give a value of 1 for a specific part-of-speech tag and 0 for all other possible tags. Or you could use the probabilistic output of a part-of-speech tagger and have values between 0 and 1 for all of the tags. Regardless, the only thing that matters is that all of our features are defined as real numbers.

Interpretation of Feature Weights

How do we interpret the impact that our features have on our model? For instance, let's revisit the essay scoring task where we have decided to choose 2 features and let's say they are essay length (word count) and number of spelling mistakes (count of words not in a dictionary). Remember that our function is:

$$\hat{y} = \beta_0 + x_1\beta_1 + x_2\beta_2$$

Here, x_1 will be essay length and x_2 will be spelling mistakes (or vice-versa). After we have fit a model, we will have values for β_0, β_1, and β_2. β_0 is simply an offset term. This is to ensure that our y values are in the correct range. β_1 tells us how important the x_1 feature is. In this example, let's assume that essays are scored from 1-5. If β_1 is positive, it means that we are adding to our score. Each additional word in an essay will add β_1 more to our score. Let's assume that misspellings hurt your score. Thus, β_2 will be negative. Each additional misspelled word would detract from your score. You can think of the predicted score, y, as being a linear combination of an offset (to make sure we are getting a score near the range we want), extra weight for additional word in an essay, and a penalty for each additional misspelled word.

Classification

Some of you may have noticed that there are no bounds on the line equation we have defined. So, when we talk about scoring an essay from 1-5

or a movie review from 0-10, we could possibly get values that are negative or even above our range. That is just an issue we need to be aware of. As a simple post-processing step we can clip the value using min and max functions to make sure we stay in the appropriate ranges. However, in general, if our training data is representative of our testing data, it should only be an issue for a very small number of cases.

Along these lines, even though linear regression predicts a real valued output, we can still use it in some classification tasks. If there is any ordinality among the classes, we can use linear regression and then map the output to a class. For instance, let's say that we have training essays that are given grades of A, B, C, and D. We can define A to be scores of 90-100, B as 80-89, C as 70-79, and D as 60-69. We then take our training data and take the midpoints of these ranges, so any A essay would be 95. Assuming our data is distributed rather evenly, or that grades average to the middle of the range (the teacher thought the work was A material on average for A grades, not that all A's were just borderline B's), we can use this as our training data. We have mapped classes (grades) into numerical scores. We can then use linear regression to predict values for testing data, with a simple post-processing step of mapping the numerical score back to a grade letter. In general, you can do this with any sort of data that is ordinal, regardless of if it is a classification task, and depending on your problem, it may actually yield better results than some classification methods.

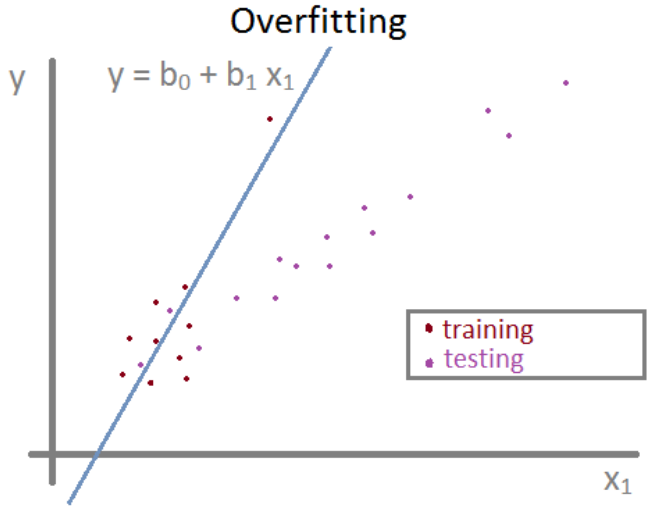
9.4 Overfitting

In general, when we are trying to learn models, we need to worry about overfitting in our data. Recall that overfitting is when we make our model fit our training data so perfectly that it does not generalize well to our testing data. Most real world data sets are noisy. If our trained model fits the data too well, we have modeled our parameters as if there was no noise, so that our model will not fit our testing data as well. There are multiple different ways to deal with the overfitting problem and regularization is one very common method of doing so.

When learning a model from data, we always have to be careful about overfitting, but the problem is particularly prevalent in Natural Language Processing. We have formally defined this problem such that we have m different features for our x values. If m gets too large relative to the n training examples we have, we will necessarily overfit our data as each parameter we learn for $\mathbf{x}$ will tend towards fitting just one of our n examples perfectly.

So, we should make sure to always choose $m << n$.

The reason the potential to overfit is so prevalent in Natural Language Processing is due to how we often choose features. For instance, in many of the methods that we have looked at so far, we choose a vocabulary size and treat each word as an independent feature. It is not uncommon to have tens of thousands of unique words in even a modest sized training corpus. If we assign each word a unique feature, our features (m) can easily outgrow the number of training examples (n). Regardless of any other ways we try to prevent overfitting, such as regularization, we must be aware of our feature set size and choose an appropriate feature set initially.



9.4.1 Regularization

We have talked about regularization earlier in this class when we discussed Logistic Regression. Regularization attempts to prevent overfitting our training data, when learning a model, by adding an additional penalty term to our objective function. This added term acts orthogonally to the first term in our objective function which is modeled on the data. There are whole classes of regularizers, but in general, they aim to impose penalties on our parameters. Often these have the goal of driving as many of our parameters to 0 (or as close to it as possible) without degrading our performance on our training data.

To implement regularization, when defining our objective function for parameter estimation, we include a regularization term. Generally, we give it a weight λ which is either given (often through trial and error) or tuned,

often with a held out set of data or cross validation. We choose a regularization term based on some desired properties. l_2 regularization is one of the most commonly used regularization methods. l_1 is also frequently chosen as a regularizer due to its property of driving many of the parameters to 0.

l_2 Regularization

l_2 regularization is simply the Euclidean distance between the origin to a point in our m dimensional space. The value of each of our parameters is squared, summed together, and finally the square root is taken. This can easily be interpreted as the distance metric commonly taught in grade school. Here's our objective function modified with the addition of an l_2 regularizer.

$$\hat{\boldsymbol{\theta}} = \underset{\theta = \langle \beta_0, \boldsymbol{\beta} \rangle}{\arg\min} \frac{1}{2n} \sum_{i=1}^{n} (y_i - (\beta_0 + \mathbf{x_i}^\intercal \boldsymbol{\beta}))^2 + \frac{\lambda}{2} \sqrt{\sum_{j=1}^{m} \beta_j^2}$$

l_1 Regularization

l_1 regularization is the Taxicab or Manhattan distance. It is the distance if you can only move along a single axis at a time. Think of it as a taxi in Manhattan that must drive down an avenue and then down a street rather than going diagonal through a block. This regularizer has the nice property of making many of our parameters go to 0. In linear regression, we have made the assumption that our features are independent. One of the intuitions behind l_1 regularization is that if two features are actually dependent, one of them will be driven to zero. Again, here is our updated objective function with l_1 regularization.

$$\hat{\boldsymbol{\theta}} = \underset{\theta = \langle \beta_0, \boldsymbol{\beta} \rangle}{\arg\min} \frac{1}{2n} \sum_{i=1}^{n} (y_i - (\beta_0 + \mathbf{x_i}^\intercal \boldsymbol{\beta}))^2 + \lambda \sum_{j=1}^{m} |\beta_j|$$

References

Bishop, Christopher M. (2009). "Pattern Recognition and Machine Learning" 8th edition. Springer Publishing.