10-315 Notes
Feature Engineering & Logistic Regression

Carnegie Mellon University
Machine Learning Department

Contents

1 Feature Engineering **1**

1.1 Intuition: Weakness of Linear Regression 1

1.2 Idea: Creating New Features in our Dataset 3

1.3 Aside: Another Perspective of Feature Engineering 4

1.4 Generalizing Example to Polynomial Features 5

1.5 Abstract Feature Transforms 6

1.6 Feature Engineering and Overfitting 7

2 Logistic Regression **8**

2.1 Logistic Regression: a Linear Model for Classification 8

2.2 Probabilistic Classification 12

2.3 New Classification Loss for Probabilities 13

1 Feature Engineering

If your dataset is complex (i.e. input values have complex relationships with output values), simpler models may be unable to appropriately model/fit to said datasets. One powerful and interesting tool is machine learning is feature engineering, which allows simpler models to fit more complex distributions of data.

1.1 Intuition: Weakness of Linear Regression

Recall our linear regression model on $x \in \mathbb{R}$

$$h(x) = wx + b$$

If we were given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ that followed a linear trend, we could easily fit a linear model, as shown below, left.

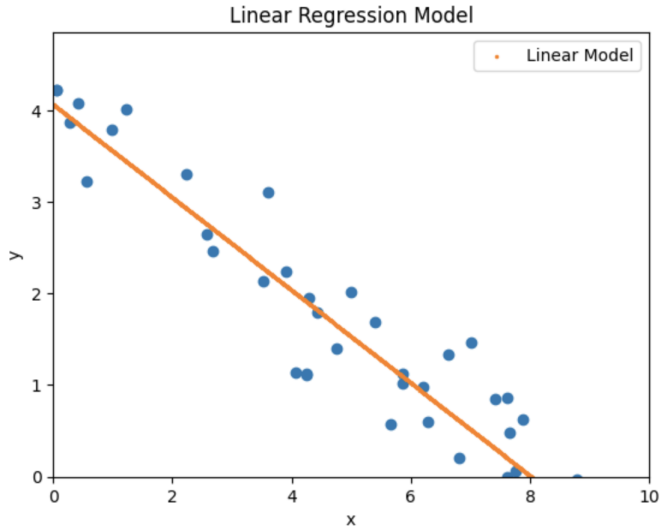


Figure 1: Linear Dataset with Linear Regression Model

The choice of the model for this dataset makes clear sense, and it fits the data well. However, suppose that our data is non-linear:

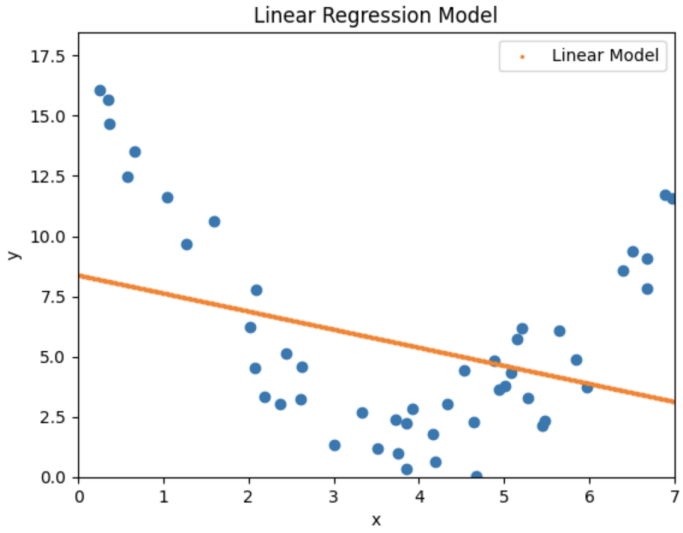


Figure 2: Linear Regression Model Fitted on Nonlinear Dataset

If we attempt to fit a linear regression model to this dataset, we can see that it doesn't necessarily model the dataset distribution well.

This is seemingly a setback for linear regression models. The linearity constraint of linear regression models makes it a model that is easily interpretable, easy to optimize, and simple; however, it would seem that it doesn't make it a good model for more complex (e.g. non-linear) distributions of data. This is where the idea of feature engineering can come into play.

1.2 Idea: Creating New Features in our Dataset

Recall Figure 2. We can see that the dataset strongly resembles a quadratic curve. For example, if we had some kind of model that learned optimal parameters $[w_1, w_2, b]^\top$ for

$$h(x) = w_1 x^2 + w_2 x + b$$

then we could create a better hypothesis for our non-linear dataset $\mathcal{D}$ because it can represent a quadratic data distribution, but how do we go about this?

While the quadratic formulation of $h(x)$ is non-linear with respect to the input data $x \in \mathbb{R}$, we can **note that $h(x)$ is still a linear function with respect to the model's parameters (w_1, w_2, b).** This means that we can apply linear regression! However, instead of just having one feature $x \in \mathbb{R}$, we have essentially constructed a new dimension/feature in our dataset $x^2 \in \mathbb{R}$ that we are using as a feature to train on, in addition to the original feature, x.

Specifically, when given the non-linear dataset $\mathcal{D}$, we can construct (or engineer) new features to create a dataset $\mathcal{D}'$ that allows us to properly model this more complex distribution.

y	x
5	6
12	7
1	3
5	2
15	0
⋮	⋮

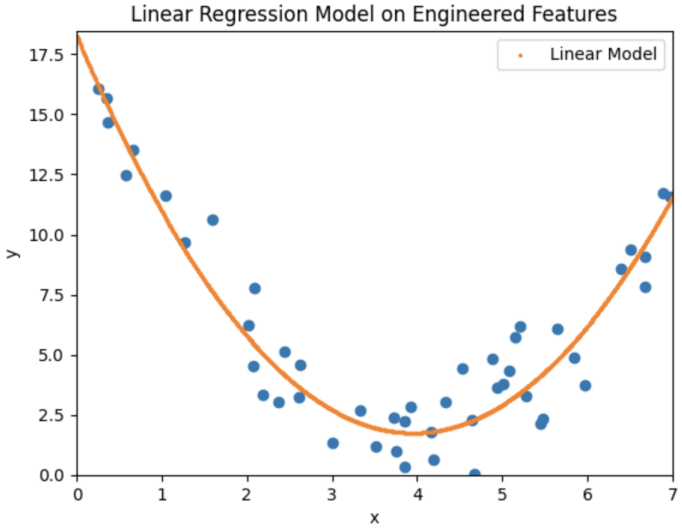
y	x	x^2
5	6	36
12	7	49
1	3	9
5	2	4
15	0	0
⋮	⋮	⋮

Table 1: Original Dataset $\mathcal{D}$ (left) and Feature-Engineered Dataset $\mathcal{D}'$ (right)

Now, we can train a linear regression model with this new dataset $\mathcal{D}'$. By adding the new feature, x^2, by hand into the dataset, we can allow our model to fit to a quadratic function. If we trained a linear regression model using the dataset $\mathcal{D}'$ where one feature is x and another feature is x^2, then we would receive a model that matches our desired quadratic hypothesis

$$h(x) = w_1 x^2 + w_2 x + b$$

and as a result, it would fit better to the dataset. Below is the result of training a linear regression model on dataset $\mathcal{D}'$:



This is extremely powerful. Note that this curve still represents the predictions of a linear regression model that takes in samples from $\mathcal{D}$, our original dataset, as input. However, when predicting on a sample, $x \in \mathbb{R}$, then we first have to engineer the feature x^2 which would then allow our model, $h(x) = w_1 x^2 + w_2 x + b$, to predict. Thus, we can see that when we train a model on a feature-engineered dataset, we have to make sure that we follow the same transformations when predicting with this model as well.

Without changing the model, we have allowed it to generalize to a new family of hypothesis functions by simply changing the dataset that we were given, $\mathcal{D}$, to a new dataset $\mathcal{D}'$. This is the primary strength that feature engineering brings to machine learning and is extremely prevalent throughout the entire field.

1.3 Aside: Another Perspective of Feature Engineering

Note that while our original dataset, $\mathcal{D}$, had only 1 feature. The linear regression model that we trained was trained on the 2 features from the engineered dataset $\mathcal{D}'$. As a result, because we are still working with 2D inputs, we can visualize the dataset that our model was trained on.

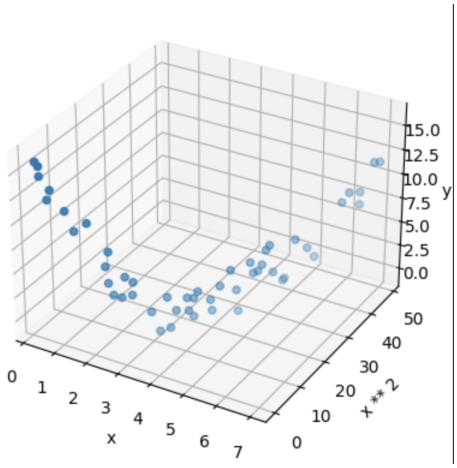


Figure 3: Feature-Engineered $\mathcal{D}'$ Plot

Because our model is trained on 2 features, instead of fitting a line to the dataset, we are now fitting a plane. We can note that this dataset $\mathcal{D}'$ can easily be fitted by a plane in this dimension. That is precisely what our newly trained model h is doing when it was trained on the engineered dataset $\mathcal{D}'$.

1.4 Generalizing Example to Polynomial Features

Note that in order to model a quadratic function, we simply engineered a new feature x^2 that allowed us to develop a model for $h(x) = w_1x^2 + w_2x + b$. However, suppose that we now wanted to model a cubic function,

$$h(x) = w_1x^3 + w_2x^2 + w_3x + b$$

or even a quartic function (degree 4 polynomial)

$$h(x) = w_1x^4 + w_2x^3 + w_3x^2 + w_4x + b$$

Then, all we have to do is simply engineer the features for the higher-powered terms, and we will be able to successfully have our h model those polynomials. In general, if we wanted to model a polynomial of degree M on our original (1-feature) dataset, $\mathcal{D}$, all we have to do is add features to construct a new dataset, as shown below.

y	x
$y^{(1)}$	$x^{(1)}$
$y^{(2)}$	$x^{(2)}$
$y^{(3)}$	$x^{(3)}$
$y^{(4)}$	$x^{(4)}$
$y^{(5)}$	$x^{(5)}$
$\vdots$	$\vdots$

y	x	x^2	x^3	$\dots$	x^M
$y^{(1)}$	$x^{(1)}$	$(x^{(1)})^2$	$(x^{(1)})^3$	$\dots$	$(x^{(1)})^M$
$y^{(2)}$	$x^{(2)}$	$(x^{(2)})^2$	$(x^{(2)})^3$	$\dots$	$(x^{(2)})^M$
$y^{(3)}$	$x^{(3)}$	$(x^{(3)})^2$	$(x^{(3)})^3$	$\dots$	$(x^{(3)})^M$
$y^{(4)}$	$x^{(4)}$	$(x^{(4)})^2$	$(x^{(4)})^3$	$\dots$	$(x^{(4)})^M$
$y^{(5)}$	$x^{(5)}$	$(x^{(5)})^2$	$(x^{(5)})^3$	$\dots$	$(x^{(5)})^M$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 2: Original Dataset $\mathcal{D}$ (left) and Feature-Engineered Dataset $\mathcal{D}'$ (right)
Notation alert: $x^{(i)}$ is the i-th sample, not an exponent!

This generalization of feature engineering to M-th degree polynomials is known as a **polynomial feature transform** where the dataset is now augmented/engineered to include polynomial terms. We will take a look at some models trained on datasets with larger degree polynomial feature transforms; however, before looking at that, we can generalize our understanding of feature engineering even further.

1.5 Abstract Feature Transforms

So far, we have looked only at feature transformations that involve exponentiating some terms inside of our dataset. However, this is just a specific case of feature engineering. Instead of applying an exponent, we could have chosen any other function we desired. We could have applied a period function sin to our feature x to construct a new feature $\sin(x)$. We could have transformed our original features by including the distance from the origin, $\|x\|_2$ as a feature as well.

In reality, we could choose any transformation that we would like to apply to our features, as long as we apply the same transformation to each sample in our dataset for training and at time of prediction.

We can represent this notion of feature transformation mathematically, if we are given a dataset $X \in \mathbb{R}^{N \times K}$ where we have N samples and K features per sample, then, we can define a feature transformation function, $\phi : \mathbb{R}^K \to \mathbb{R}^M$, where M is the number of features that are produced by the transformation ϕ. Now, when we want to train our model on our feature-transformed dataset, we can write the notation as follows, $\phi(X)$, where:

$$X = \begin{bmatrix} — \, \mathbf{x}^{(1)^\top} \, — \\ — \, \mathbf{x}^{(2)^\top} \, — \\ — \, \mathbf{x}^{(3)^\top} \, — \\ \cdots \\ — \, \mathbf{x}^{(N)^\top} \, — \end{bmatrix} \qquad \phi(X) = \begin{bmatrix} — \, \phi\left(\mathbf{x}^{(1)}\right)^\top \, — \\ — \, \phi\left(\mathbf{x}^{(2)}\right)^\top \, — \\ — \, \phi\left(\mathbf{x}^{(3)}\right)^\top \, — \\ \cdots \\ — \, \phi\left(\mathbf{x}^{(N)}\right)^\top \, — \end{bmatrix}$$

Note, that now, the dataset matrix $\phi(X) \in \mathbb{R}^{N \times M}$ because we've applied the feature transform ϕ to each sample $\mathbf{x}^{(i)}$ in our dataset. We now have a generalized notion of feature transformations as this function ϕ which machine learning engineers (that's you!) can define to be anything.

Example 1: Relating back to our original example, when performing a polynomial feature transformation of degree 3, we were using the feature transform

$$\phi\left([x]\right) = [x, x^2, x^3]^\top$$

Example 2: Another example of feature transformations that we have been implicitly using is the feature transform that appends the bias coefficient of 1 inside of our design matrix for linear regression models:

$$\phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}$$

allowing us to model linear models that don't necessarily go through the origin (i.e., affine models). It is useful to think about how this feature transform allows us to create the design matrix used in linear regression.

1.6 Feature Engineering and Overfitting

Before concluding this chapter on an introduction to feature engineering, one might be asking, "why not just have more and more features?". In the examples above, we've shown how adding more features can allow us to create models that are able to fit to more complex data distributions, such as k-dimensional polynomial curves.

However, if adding more features allows our model to generalize to a larger family of functions, why not add as many features as we can? What downside is there to having too many features? Unfortunately, there are some downsides to applying high-dimensional and complex transformations to your dataset. To show this, we will look at what happens to a linear regression model when we apply polynomial feature transforms of degree k for increasing k on a linear dataset, $\mathcal{D}$.

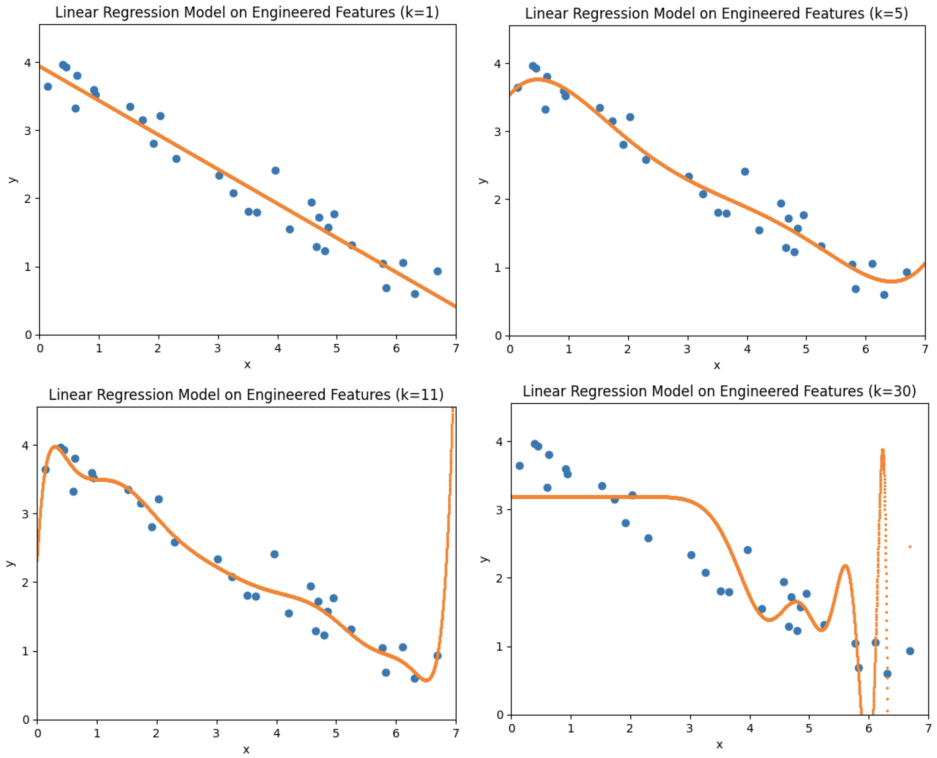


Figure 4: Linear regression with $k = \{1, 5, 11, 30\}$ polynomial feature transforms

As we can see, when we increase the number of features to be very large, the model starts to break down and begins to prioritize matching the exact distribution of the training data as opposed to capturing its general trend. This is a prime example of model overfitting and can commonly occur when the dimensionality of the data is too high, which is the case with complex feature transforms. As a result, we can note that it may not be best to always have very large feature transformations. Instead, we must make reasonable transformations that both improve the model's ability to generalize well to the data distribution without allowing it to overfit.

There's certainly more to come in this course on how to combat overfitting despite having very expressive models, such as linear regression with high-degree polynomial features and even neural networks.

2 Logistic Regression

On a different topic, we are going to take a look at classification from a new perspective. Previously, we have viewed classification as making an explicit decision on whether a given sample, $\mathbf{x}^{(i)}$, belongs to one of K classes. However, making a discrete decision on the class of $\mathbf{x}^{(i)}$ doesn't necessarily provide information on how strongly we believe $\mathbf{x}^{(i)}$ belongs to a given class. This idea will raise the notion of developing a *probability distribution* over classes when performing prediction.

Additionally, some classification models that we have investigated are the Decision Tree and K-nearest neighbor models. However, we can note that these models aren't necessarily linear functions with respect to their input, $\mathbf{x}^{(i)}$. Instead, they perform some non-linear function in order to decide their classes. This is where logistic regression models arise. Logistic regression models are an intuitive extension to our well-known linear regression models that allow us to perform classification in a linear manner.

2.1 Logistic Regression: a Linear Model for Classification

Now that we have covered this new intuition for performing classification in a probabilistic manner, we can look at a new model that performs this kind of behavior, the logistic regression model. We will inspire an understanding of this model with an example.

Suppose that we wanted to perform classification on a dataset $\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1}^{N}$ where $x^{(i)} \in \mathbb{R}$ and $y^{(i)} \in \{0, 1\}$ and that we wanted to create a linear classifier (i.e. the decision boundary is linear).

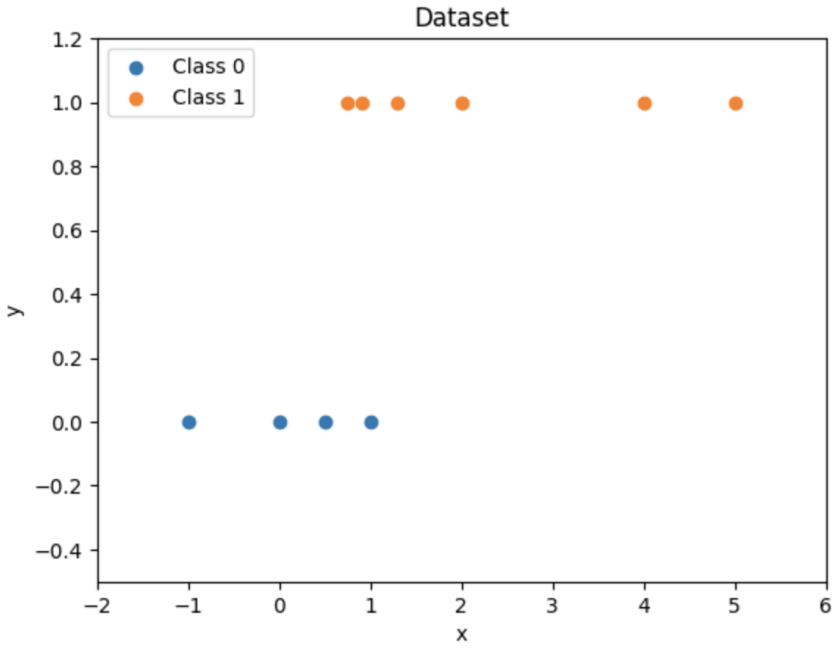


Figure 5: Dataset to Train Classifier On

One idea that we might have to implement this linear classifier is to use a simple linear regression model where $\hat{y} = wx + b$. Training such a model produces the following regression.

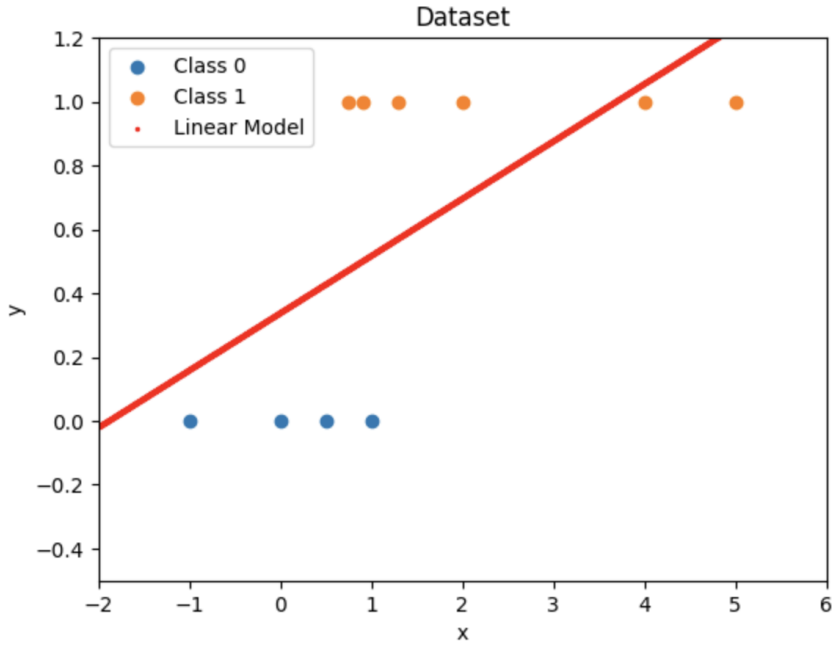


Figure 6: $\mathcal{D}$ trained on Linear Regression Model

We can see that our model learns some relationship between the input x and the class y, where the class will increase if the input increases. However, this doesn't necessarily constrain the output of a model to the appropriate class that we would like, either a 0 or a 1. As a result, we can consider placing some threshold on this linear regression. For example, at a value of 0.5, we can threshold our function to class 0 if the linear regression is less than 0.5 and to class 1 otherwise. This thresholding produces a new model formulation $\hat{y} = g_{threshold}(wx + b)$. Training such a model produces a fit as follows

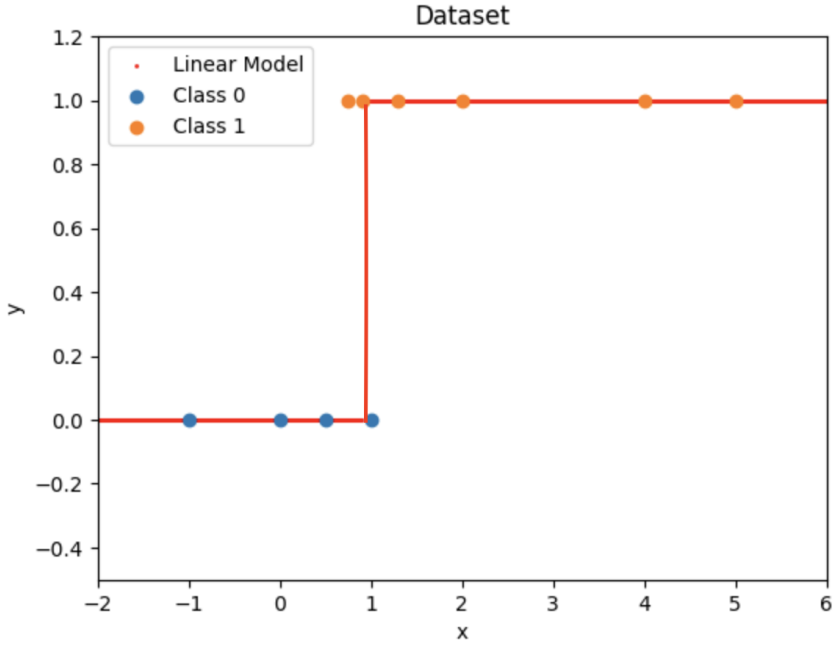


Figure 7: $\mathcal{D}$ trained on thresholding model

While our model now produces only valid values of $\hat{y}$, we still have an issue described earlier in these

notes. The model is extremely certain about its predictions! It transitions rapidly from one class to the other right around input values $x = 1$. However, looking at the overlap in our data, we probably want our model to state that for $x = 1$, $P(Y = 1 \mid x) \approx 0.5$. In addition to this hard-classification property of the threshold model, it has a derivative of zero essentially everywhere, making it very problematic to optimize over.

As a result, we can try working with a different model $h(x)$. Note, that in order to create a valid classifier, we simply applied the threshold function $g_{threshold}$ onto a linear combination of the features $wx + b$. We can apply the same idea but with a function that smoothly maps all real input values $(-\infty, \infty)$ to the range $(0, 1)$ (i.e., a probability distribution for binary classification). One such common function is a specific sigmoid (s-shaped) function, the **logistic function**, which is often just called the **sigmoid function** and denoted as σ:

$$g_{logistic}(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

A plot of this function is in Figure 8. Prove to yourself that this function outputs values in $[0, 1]$ and outputs 0.5 when the input is 0.

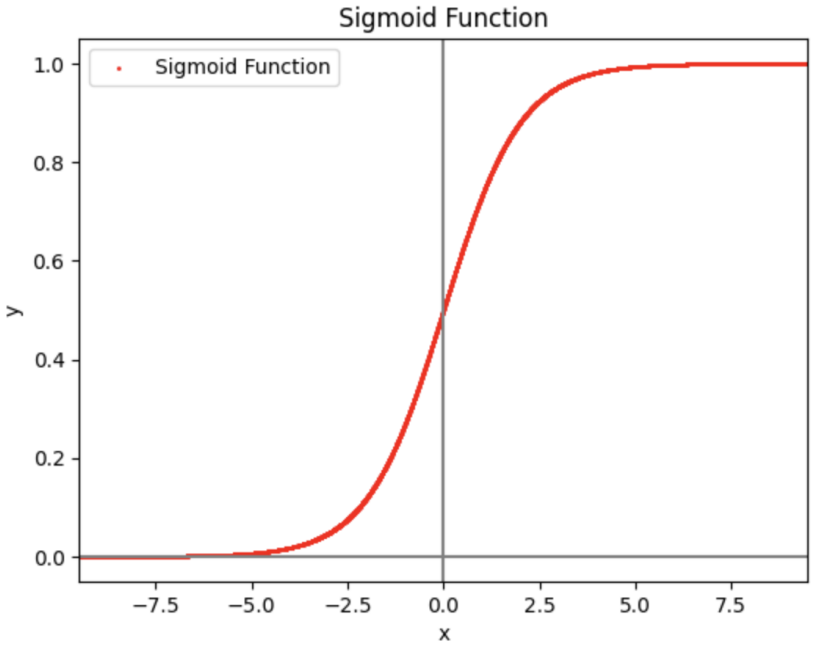


Figure 8: Plot of logistic (sigmoid) function

Now, with the logistic function, $\sigma(z)$, we can apply it to our linear transformation of our input, $wx + b$, to get a new model

$$h(x) = \hat{y} = \sigma(wx + b)$$

This is the logistic regression model for 1-dimensional inputs. If we wanted to generalize this model to samples $\mathbf{x} \in \mathbb{R}^M$, where we have M features (assume the bias term is included in this $\mathbf{x}$), then a more general definition of the logistic regression model would be:

$$h(x) = \hat{y} = \sigma(\mathbf{w}^\top \mathbf{x})$$

Note that this model is used for performing binary classification. Because the output of the model $h(x)$ is a value in $(0, 1)$, we can re-think this value as a probability where $h(x) = P(Y = 1 \mid x)$. We can read this as our logistic regression model producing the probability of a sample, x, belonging to class 1.

Finally, after defining a new logistic regression model, $h(x)$, we can optimize over it on our toy dataset $\mathcal{D}$, see Figure 9.

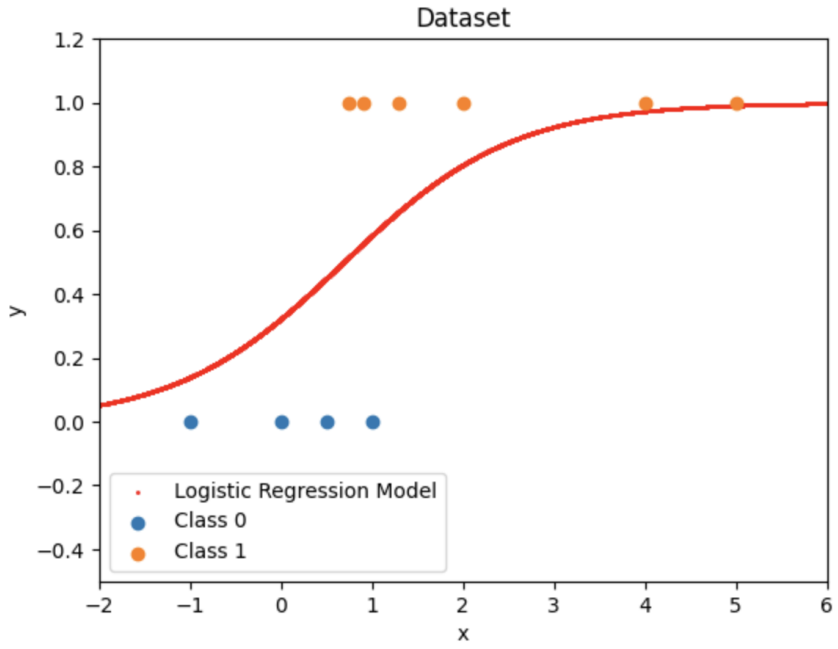


Figure 9: $\mathcal{D}$ trained on logistic regression model

We can now see that our logistic regression model is a prediction function that accounts for uncertainty in our data and indicates a probability for a sample being in class 1. While this example may seem simple, the logistic regression model is an extremely powerful model. It is used as a building block for many larger machine learning models, can be learned efficiently, and is great for simple classification problems.

In lecture, we will investigate this model further by looking at how we can estimate the error of this model on a dataset, how we can use that information to train our model, and how we can apply this model to multi-class classification problems.

2.2 Probabilistic Classification

For some motivation, suppose that we are predicting whether it is going to rain today using some features like humidity, cloud coverage, etc. In the past, we have used models that firmly decide which class a sample will belong to. For example, if we are given a sample, $\mathbf{x}$, and we have to assign it to one of K classes. Our past models, such as the decision tree and k-nearest neighbor models, would make a decision (typically through majority vote) and say that $\mathbf{x}$ is assigned label k. However, this effectively states that our model believes that $\mathbf{x}$ has a 100% chance of belonging to class with label k and a 0% chance with all other labels.

This can be problematic when our models are very uncertain about their predictions. Let's say we were predicting the type of a flower given its sepal width and sepal length in the classic iris classification dataset. Typically, we would either make a decision and say that a given sample is one of 3 classes, a *setosa*, a *versicolor*, or *virginica*. But if we were uncertain, it wouldn't make sense to decidedly state which flower our sample is. Instead, it would be beneficial if we could estimate how confident we are in our predictions. For example, our model could say that there is a 25% chance of the flower being a *setosa*, a 25% chance of the flower being a *versicolor*, and a 50% chance of our flower being a *virginica*. Allowing our model to produce probabilities is useful for portraying uncertainty in our model.

In Figure 10, we can see that we are trying to classify iris samples in two fundamentally different ways. One uses K-nearest neighbor to give a hard classification of a single class to each possible input, x_1, x_2, whereas the other figure uses logistic regression to provide a soft classification by providing the probability of the sample being in each of the three classes.

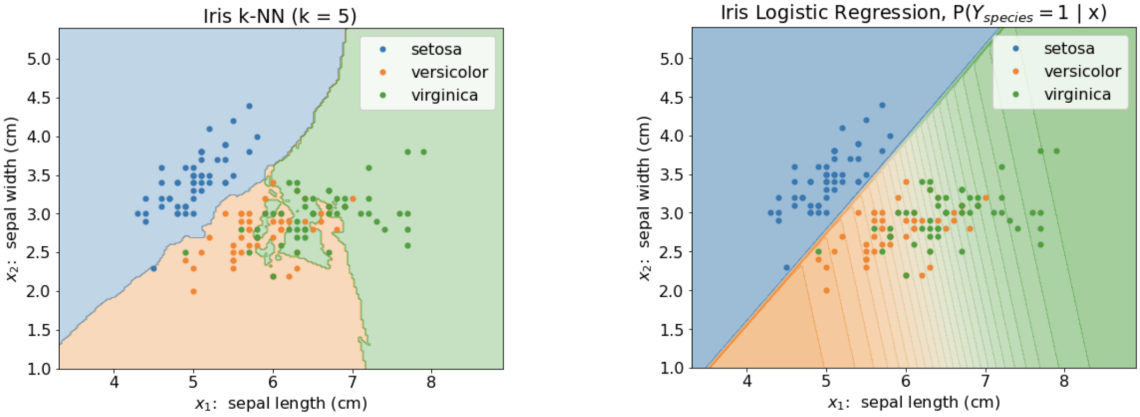


Figure 10: Hard classification vs. soft (probabilistic) classification. The logistic regression plot (right) shows contour lines colored for each class as the probabilities for those classes move from 0.0 (transparent white) to 1.0 (solid blue/orange/green). The logistic regression plot shows a gradual transition from solid orange to solid green, particularly in the region where there is considerable overlap between sepal features for *versicolor* (orange) and *virginica* (green).

In the plot on the left, we can see that the k-NN model has very strict decision boundaries where if a sample lands in a region belonging to a class, the k-NN model will predict the sample as that class with 100% confidence. However, when attempting to classify a point like $\mathbf{x} = [x_1, x_2] = [6.1, 3]$, we can see that there is a relatively even mix of both *versicolor* and *virginica* classes around that point. As a result, it would be better if the model stated that there is a 0% chance of $\mathbf{x}$ being a *setosa*, a 50% chance of $\mathbf{x}$ being a *versicolor*, and a 50% chance of $\mathbf{x}$ being a *virginica*.

This is what the model on the right is doing. This logistic regression classification model (which we will learn about soon), indicates the probability $P(Y_{\text{species}} = 1 \mid \mathbf{x})$. As a result, when given the sample, $\mathbf{x} = [x_1, x_2] = [6.1, 3]$, instead of producing a hard classification, the model produces a vector, $\hat{\mathbf{y}}$, that represents a probability distribution. Let h be our logistic regression prediction

function for input $\mathbf{x}$. Then the predicted output would have the following vector form:

$$\hat{\mathbf{y}} = h(\mathbf{x}) = \begin{bmatrix} P(Y_{setosa} = 1 \mid \mathbf{x}) \\ P(Y_{versicolor} = 1 \mid \mathbf{x}) \\ P(Y_{virginica} = 1 \mid \mathbf{x}) \end{bmatrix} = \begin{bmatrix} 0 \\ 0.5 \\ 0.5 \end{bmatrix}$$

instead of something like $\hat{y} = h(\mathbf{x}) = 2$ where 2 represents the class *virginica*. This is the goal of probabilistic classification. Instead of having a model that decides what class a sample belongs to, the model will estimate the probability that a sample belongs to a class. As a result, if we had a probabilistic classification model when predicting on a sample, $\mathbf{x}$, it produces a probability distribution over the classes, $\hat{\mathbf{y}}$, where $\hat{y}_k = P(Y_k = 1 \mid \mathbf{x})$ where Y_k is the random variable of class k being the correct class.

One point to note is that even if our model produces probabilities for a sample belonging to a class, we can still produce a decision (or assignment) on which class a sample belongs to. The typical policy for this is to assign a sample $\mathbf{x}$ that has probability distribution $\hat{\mathbf{y}}$ over K classes the label with the maximum probability. In other words, the label we assign to that sample is $\text{argmax}_{k \in \{1,\ldots,K\}} \hat{y}_k$.

2.3 New Classification Loss for Probabilities

We now have a new formulation for how we can do classification. Instead of producing hard labels, we produce a probability distribution over the classes, representing the probability of a sample belonging to a class. However, with a new representation of our model output, $\hat{\mathbf{y}}$, we now need to have a new loss function.

Originally, when performing decision-based predictions, our ground-truth label, $y \in \mathcal{C}$ would be a label indicating the class of the sample, and our model would output one of those labels, $\hat{y} \in \mathcal{C}$ where $\mathcal{C} = \{0, 1, 2\}$ (each number corresponds to a class). With this representation, we defined a loss function that was effectively an indicator function

$$\ell(y, \hat{y}) = \mathbb{1}[\hat{y} \neq y]$$

However, our representation of the targets has changed because we would now like to predict a probability distribution. Because of this, we will have to map our ground-truth labels, y to probability distributions as well. If our sample belonged to class k in our dataset, then the associated sample, $\mathbf{y}$, is now an element of $\mathbb{R}^K$ where we have K classes (note it is a vector). All elements of $\mathbf{y}$ will be 0 except for the k-th element, y_k, which will be 1. This is known as a **one-hot vector**, and it describes a probability distribution that the sample belongs to the k-th class with probability 1 and 0 probability with all other classes.

In the iris classification example above, samples belonging to class 1 (*versicolor*, we are 0-indexing) would have their labels as follows.

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

As shown earlier, our model outputs probability distributions. Suppose for the label $\mathbf{y}$ above, we have the prediction

$$\hat{\mathbf{y}} = \begin{bmatrix} 0.15 \\ 0.7 \\ 0.15 \end{bmatrix}$$

Recall, that we need to now define a new loss function $\ell(\mathbf{y}, \hat{\mathbf{y}})$ because our original indicator function definition on target labels won't work. If we take a look at a plot of these two vectors (in this case, probability distributions), we can see that there is some difference between them that we could quantify.

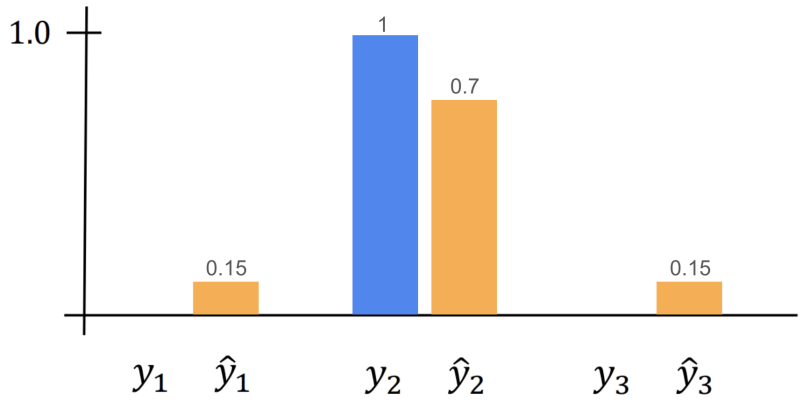


Figure 11: Plot of probability distributions defined for the true output value $\mathbf{y}$ (blue) and the predicted output $\hat{\mathbf{y}}$ (orange). Note that the true output will always be a **one-hot vector** (vector with exactly one 1 value), while the predicted output may have some uncertainty about which class the sample belongs to.

One common loss function when performing this form of classification is called **cross-entropy loss**. The cross-entropy loss is generally a function that compares any two probability distributions. In our case, this is perfect because we would like to compare the probability distribution that our model produces, $\hat{\mathbf{y}}$, with the probability distribution defined by our ground truth label, $\mathbf{y}$. Given that K is the number of classes that we are predicting over, the cross-entropy loss definition is as follows:

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

For the example above with $\mathbf{y} = [0, 1, 0]^\top$ and $\hat{\mathbf{y}} = [0.15, 0.7, 0.15]^\top$, the cross-entropy loss is:

$$\begin{aligned} \ell(\mathbf{y}, \hat{\mathbf{y}}) = & \quad - y_1 \log \hat{y}_1 \quad - y_2 \log \hat{y}_2 - y_3 \log \hat{y}_3 \\ = & \quad - 0 \log 0.15 - 1 \log 0.7 - 0 \log 0.15 \\ = & \quad 0.36 \end{aligned}$$

whereas if the prediction was perfect, $\hat{\mathbf{y}} = [0.0, 1.0, 0.0]^\top$, the cross-entropy would be 0.0:

$$\begin{aligned} \ell(\mathbf{y}, \hat{\mathbf{y}}) = & \quad -0 \log 0.0 - 1 \log 1.0 - 0 \log 0.0 \\ = & \quad 0.0 \end{aligned}$$

Note that, by convention in machine learning, we use the natural log to compute cross-entropy values.

In future material, we will go over how this loss is used in various machine learning models, how we use it for training, and some more interesting properties of this loss function. As an exercise for gaining a better understanding of cross-entropy loss, it is useful to take various probability distributions (e.g. distributions with $K = 3$ classes) and calculate the cross-entropy loss between them. Understand what terms in the summation cause the loss to grow or not grow.