CS 188 Introduction to Artificial Intelligence
Fall 2022 Note 20

These lecture notes are based on notes originally written by Nikhil Sharma and on the textbook *Artificial Intelligence: A Modern Approach.*

Last updated: November 8, 2022

Linear Regression

Now we'll move on from our previous discussion of Naive Bayes to **Linear Regression**. This method, also called **least squares**, dates all the way back to Carl Friedrich Gauss and is one of the most studied tools in machine learning and econometrics.

Regression problems are a form of machine learning problem in which the output is a continuous variable (denoted with y). The features can be either continuous or categorical. We will denote a set of features with $\mathbf{x} \in \mathbb{R}^n$ for n features, i.e. $\mathbf{x} = (x_1, \dots, x_n)$.

We use the following linear model to predict the output:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \cdots w_n x_n$$

where the weights w_i of the linear model are what we want to estimate. The weight w_0 corresponds to the intercept of the model. Sometimes in literature we add a 1 on the feature vector $\mathbf{x}$ so that we can write the linear model as $\mathbf{w}^T\mathbf{x}$ where now $\mathbf{x} \in \mathbb{R}^{n+1}$. To train the model, we need a metric of how well our model predicts the output. For that we will use the $L2$ loss function which penalizes the difference of the predicted from the actual output using the $L2$ norm. If our training dataset has N data points then the loss function is defined as follows:

$$Loss(h_{\mathbf{w}}) = \frac{1}{2}\sum_{j=1}^{N} L2(y_j, h_{\mathbf{w}}(\mathbf{x}_j)) = \frac{1}{2}\sum_{j=1}^{N}(y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2 = \frac{1}{2}\left\|\mathbf{y} - \mathbf{X}\mathbf{w}\right\|_2^2$$

Note that $\mathbf{x}_j$ corresponds to the jth data point $\mathbf{x}_j \in \mathbb{R}^n$. The term $\frac{1}{2}$ is just added to simplify the expressions of the closed form solution. The last expression is an equivalent formulation of the loss function which makes the least square derivation easier. The quantities $\mathbf{y}, \mathbf{X}$ and $\mathbf{w}$ are defined as follows:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_1^1 & \cdots & x_n^1 \\ 1 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_1^N & \cdots & x_n^N \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix},$$

where $\mathbf{y}$ is the vector of the stacked outputs, $\mathbf{X}$ is the matrix of the feature vectors where x_i^j denotes the ith component of the jth data point. The least squares solution denoted with $\hat{\mathbf{w}}$ can now be derived using basic linear algebra rules [1]. More specifically, we will find the $\hat{\mathbf{w}}$ that minimizes the loss function by differentiating the loss function and setting the derivative equal to zero.

$$\nabla_{\mathbf{w}}\frac{1}{2}\|\mathbf{y}-\mathbf{X}\mathbf{w}\|_2^2 = \nabla_{\mathbf{w}}\frac{1}{2}\left(\mathbf{y}-\mathbf{X}\mathbf{w}\right)^T\left(\mathbf{y}-\mathbf{X}\mathbf{w}\right) = \nabla_{\mathbf{w}}\frac{1}{2}\left(\mathbf{y}^T\mathbf{y}-\mathbf{y}^T\mathbf{X}\mathbf{w}-\mathbf{w}^T\mathbf{X}^T\mathbf{y}+\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\right)$$
$$= \nabla_{\mathbf{w}}\frac{1}{2}\left(\mathbf{y}^T\mathbf{y}-2\mathbf{w}^T\mathbf{X}^T\mathbf{y}+\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\right) = -\mathbf{X}^T\mathbf{y}+\mathbf{X}^T\mathbf{X}\mathbf{w}.$$

Setting the gradient equal to zero we obtain:

$$-\mathbf{X}^T\mathbf{y}+\mathbf{X}^T\mathbf{X}\mathbf{w} = 0 \Rightarrow \hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

Having obtained the estimated vector of weights we can now make a prediction on new unseen test data points as follows:

$$h_{\hat{\mathbf{w}}}(\mathbf{x}) = \hat{\mathbf{w}}^T\mathbf{x}.$$

Perceptron

Linear Classifiers

The core idea behind Naive Bayes is to extract certain attributes of the training data called features and then estimate the probability of a label given the features: $P(y|f_1, f_2, ...f_n)$. Thus, given a new data point, we can then extract the corresponding features, and classify the new data point with the label with the highest probability given the features. This all, however, this requires us to estimate distributions, which we did with MLE. What if instead we decided not to estimate the probability distribution? Lets start by looking at a simple linear classifier, which we can use for **binary classification**, which is when the label has two possibilities, positive or negative.

The basic idea of a **linear classifier** is to do classification using a linear combination of the features– a value which we call the **activation**. Concretely, the activation function takes in a data point, multiplies each feature of our data point, $f_i(\mathbf{x})$, by a corresponding weight, w_i, and outputs the sum of all the resulting values. In vector form, we can also write this as a dot product of our weights as a vector, $\mathbf{w}$, and our featurized data point as a vector $\mathbf{f}(\mathbf{x})$:

$$\text{activation}_w(\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) = \sum_i w_i f_i(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) = \mathbf{w}\cdot\mathbf{f}(\mathbf{x})$$

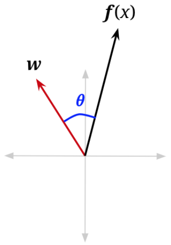
How does one do classification using the activation? For binary classification, when the activation of a data point is positive, we classify that data point with the positive label, and if it is negative, we classify with the negative label.

$$\text{classify}(\mathbf{x}) = \begin{cases} + & \text{if } h_{\mathbf{w}}(\mathbf{x}) > 0 \\ - & \text{if } h_{\mathbf{w}}(\mathbf{x}) < 0 \end{cases}$$

To understand this geometrically, let us reexamine the vectorized activation function. We can rewrite the dot product as follows, where $||\cdot||$ is the magnitude operator and θ is the angle between $\mathbf{w}$ and $\mathbf{f}(\mathbf{x})$:

[1] For matrix algebra rules you can refer to The Matrix Cookbook.

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{f}(\mathbf{x}) = \|\mathbf{w}\| \|\mathbf{f}(\mathbf{x})\| \cos(\theta)$$



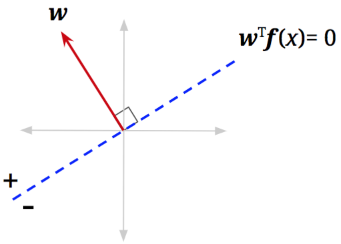
Since magnitudes are always non-negative, and our classification rule looks at the sign of the activation, the only term that matters for determining the class is $\cos(\theta)$.

$$\text{classify}(\mathbf{x}) = \begin{cases} + & \text{if } cos(\theta) > 0 \\ - & \text{if } cos(\theta) < 0 \end{cases}$$

We, therefore, are interested in when $cos(\theta)$ is negative or positive. It is easily seen that for $\theta < \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $(0,1]$, which is positive. For $\theta > \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $[-1,0)$, which is negative. You can confirm this by looking at a unit circle. Essentially, our simple linear classifier is checking to see if the feature vector of a new data point roughly "points" in the same direction as a predefined weight vector and applies a positive label if it does.

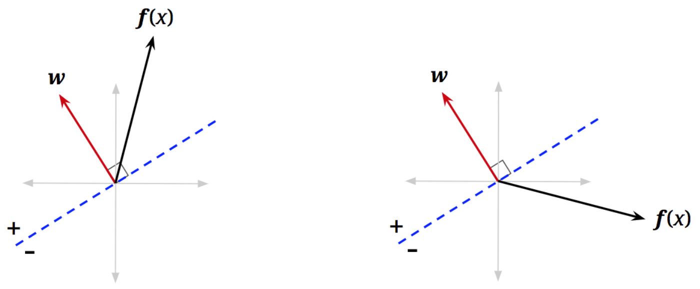
$$\text{classify}(\mathbf{x}) = \begin{cases} + & \text{if } \theta < \frac{\pi}{2} \qquad \text{(i.e. when } \theta \text{ is less than } 90°\text{, or acute)} \\ - & \text{if } \theta > \frac{\pi}{2} \qquad \text{(i.e. when } \theta \text{ is greater than } 90°\text{, or obtuse)} \end{cases}$$

Up to this point, we haven't considered the points where activation$_w(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) = 0$. Following all the same logic, we will see that $cos(\theta) = 0$ for those points. Furthermore, $\theta = \frac{\pi}{2}$ (i.e θ is 90°) for those points. In otherwords, these are the data points with feature vectors that are orthogonal to $\mathbf{w}$. We can add a dotted blue line, orthogonal to $\mathbf{w}$, where any feature vector that lies on this line will have activation equaling 0.



Decision Boundary

We call this blue line the **decision boundary** because it is the boundary that separates the region where we classify data points as positive from the region of negatives. In higher dimensions, a linear decision boundary is generically called a **hyperplane**. A hyperplane is a linear surface that is one dimension lower than the latent space, thus dividing the surface in two. For general classifiers (non-linear ones), the decision boundary may not be linear, but is simply defined as surface in the space of feature vectors that separates the classes. To classify points that end up on the decision boundary, we can apply either label since both classes are equally valid (in the algorithms below, we'll classify points on the line as positive).



x **classified into positive class** *x* **classified into negative class**

Binary Perceptron

Great, now you know how linear classifiers work, but how do we build a good one? When building a classifier, you start with data, which are labeled with the correct class, we call this the **training set**. You build a classifier by evaluating it on the training data, comparing that to you training labels, and adjusting the parameters of your classifier until you reach your goal.

Let's explore one specific implementation of a simple linear classifier: the binary perceptron. The perceptron is a binary classifier–though it can be extended to work on more than two classes. The goal of the binary perceptron is to find a decision boundary that perfectly separates the training data. In other words, we're seeking the best possible weights– the best **w**– such that any featured training point that is multiplied by the weights, can be perfectly classified.

The Algorithm

The perceptron algorithm works as follows:

1. Initialize all weights to 0: $\mathbf{w} = \mathbf{0}$
2. For each training sample, with features $\mathbf{f}(\mathbf{x})$ and true class label $y^* \in \{-1, +1\}$, do:
   (a) Classify the sample using the current weights, let y be the class predicted by your current $\mathbf{w}$:
   $$y = \text{classify}(x) = \begin{cases} +1 & \text{if } h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) > 0 \\ -1 & \text{if } h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) < 0 \end{cases}$$
   (b) Compare the predicted label y to the true label y^*:
   - If $y = y^*$, do nothing
   - Otherwise, if $y \neq y^*$, then update your weights: $\mathbf{w} \leftarrow \mathbf{w} + y^*\mathbf{f}(\mathbf{x})$
3. If you went through **every** training sample without having to update your weights (all samples predicted correctly), then terminate. Else, repeat step 2

Updating weights

Let's examine and justify the procedure for updating our weights. Recall that in step 2b above, when our classifier is right, nothing changes. But when our classifier is wrong, the weight vector is updated as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + y^*\mathbf{f}(\mathbf{x})$$

where y^* is the true label, which is either 1 or -1, and $\mathbf{x}$ is the training sample which we mis-classified. You can interpret this update rule to be:

<u>**Case 1 :**</u> mis-classified positive as negative $\quad \mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(\mathbf{x})$

<u>**Case 2 :**</u> mis-classified negative as positve $\quad \mathbf{w} \leftarrow \mathbf{w} - \mathbf{f}(\mathbf{x})$

Why does this work? One way to look at this is to see it as a balancing act. Mis-classification happens either when the activation for a training sample is much smaller than it should be (causes a Case 1 misclassification) or much larger than it should be (causes a Case 2 misclassification).

Consider Case 1, where activation is negative when it should be positive. In other words, the activation is too small. How we adjust $\mathbf{w}$ should strive to fix that and make the activation larger for that training sample. To convince yourself that our update rule $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(\mathbf{x})$ does that, let us update $\mathbf{w}$ and see how the activation changes.

$$h_{\mathbf{w}+\mathbf{f}(\mathbf{x})}(\mathbf{x}) = (\mathbf{w} + \mathbf{f}(x))^T \mathbf{f}(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{x})^T \mathbf{f}(\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) + \mathbf{f}(\mathbf{x})^T \mathbf{f}(\mathbf{x})$$

Using our update rule, we see that the new activation increases by $\mathbf{f}(\mathbf{x})^T \mathbf{f}(\mathbf{x})$, which is a postive number, therefore showing that our update makes sense. Activation is getting larger– closer to becoming positive. You can repeat the same logic for when the classifier is mis-classifying because the activation is too large (activation is positive when it should be negative). You'll see that the update will cause the new activation to decrease by $\mathbf{f}(\mathbf{x})^T \mathbf{f}(\mathbf{x})$, thus getting smaller and closer to classifying correctly.

While this makes it clear why we are adding and subtracting *something*, why would we want to add and subtract our sample point's features? A good way to think about it, is that the weights aren't the only thing that determines this score. The score is determined by multiplying the weights by the relevant sample. This means that certain parts of a sample contribute more than others. Consider the following situation where x is a training sample we are given with true label $y^* = -1$:

$$\mathbf{w}^T = \begin{bmatrix} 2 & 2 & 2 \end{bmatrix}, \mathbf{f}(\mathbf{x}) = \begin{bmatrix} 4 \\ 0 \\ 1 \end{bmatrix} \qquad h_{\mathbf{w}}(\mathbf{x}) = (2 * 4) + (2 * 0) + (2 * 1) = 10$$

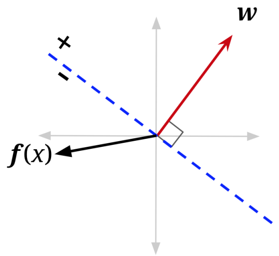
We know that our weights need to be smaller because activation needs to be negative to classify correctly. We don't want to change them all the same amount though. You'll notice that the first element of our sample, the 4, contributed much more to our score of 10 than the third element, and that the second element did not contribute at all. An appropriate weight update, then, would change the first weight alot, the third weight a little, and the second weight should not be changed at all. After all, the second and third weights might not even be that broken, and we don't fix want to fix what isn't broken!

When thinking about a good way to change our weight vector in order to fulfill the above desires, it turns out just using the sample itself does in fact do what we want; it changes the first weight by a lot, the third weight by a little, and doesn't change the second weight at all!

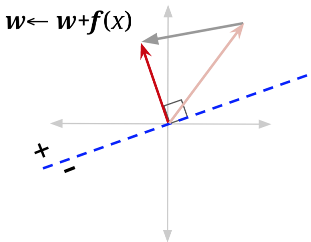
A visualization may also help. In the figure below, $\mathbf{f}(\mathbf{x})$ is the feature vector for a data point with positive class ($y^* = +1$) that is currently misclassified – it lies on the wrong side of the decision boundary defined by "old $\mathbf{w}$". Adding it to the weight vector produces a new weight vector which has a smaller angle to $\mathbf{f}(\mathbf{x})$. It also shifts the decision boundary. In this example, it has shifted the decision boundary enough so that x will now be correctly classified (note that the mistake won't always be fixed – it depends on the size of the weight vector, and how far over the boundary $\mathbf{f}(\mathbf{x})$ currently is).

Bias

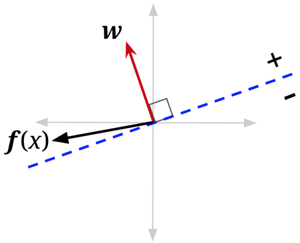
If you tried to implement a perceptron based on what has been mentioned thus far, you will notice one particularly unfriendly quirk. Any decision boundary that you end up drawing will be crossing the origin. Basically, your perceptron can only produce a decision boundary that could be represented by the function $\mathbf{w}^\top \mathbf{f}(\mathbf{x}) = 0$, $\mathbf{w}, \mathbf{f}(\mathbf{x}) \in \mathbb{R}^n$. The problem is, even among problems where there is a linear decision boundary



Mis-classifying x with old w



Updating w

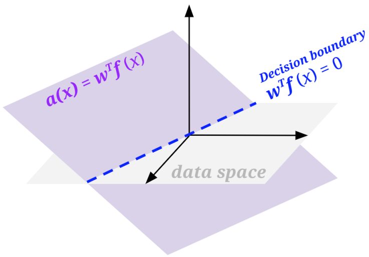


Updated classification of x

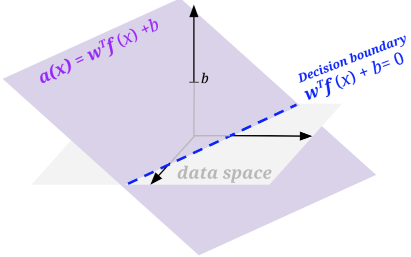
that separates the positive and negative classes in the data, that boundary may not go through the origin, and we want to be able to draw those lines.

To do so, we will modify our feature and weights to add a bias term: add a feature to your sample feature vectors that is always 1, and add an extra weight for this feature to your weight vector. Doing so essentially allows us to produce a decision boundary representable by $\mathbf{w}^\top\mathbf{f}(\mathbf{x}) + b = 0$, where b is the weighted bias term (i.e. 1 * the last weight in the weight vector).

Geometrically, we can visualize this by thinking about what the activation function looks like when it is $\mathbf{w}^\top\mathbf{f}(\mathbf{x})$ and when there is a bias $\mathbf{w}^\top\mathbf{f}(\mathbf{x}) + b$. To do so, we need to be one dimension higher than the space of our featurized data (labeled data space in the figures below). In all the above sections, we had only been looking at a flat view of the data space.



Without bias



With bias

Example

Let's see an example of running the perceptron algorithm step by step.

Let's run one pass through the data with the perceptron algorithm, taking each data point in order. We'll start with the weight vector $[w_0, w_1, w_2] = [-1, 0, 0]$ (where w_0 is the weight for our bias feature, which remember is always 1).

Training Set

#	f_1	f_2	y^*
1	1	1	-
2	3	2	+
3	2	4	+
4	3	4	+
5	2	3	-

Single Perceptron Update Pass

step	Weights	Score	Correct?	Update
1	[-1, 0, 0]	$-1\cdot1+0\cdot1+0\cdot1=-1$	yes	none
2	[-1, 0, 0]	$-1\cdot1+0\cdot3+0\cdot2=-1$	no	+[1, 3, 2]
3	[0, 3, 2]	$0\cdot1+3\cdot2+2\cdot4=14$	yes	none
4	[0, 3, 2]	$0\cdot1+3\cdot3+2\cdot4=17$	yes	none
5	[0, 3, 2]	$0\cdot1+3\cdot2+2\cdot3=12$	no	-[1, 2, 3]
6	[-1, 1, -1]			

We'll stop here, but in actuality this algorithm would run for many more passes through the data before all the data points are classified correctly in a single pass.

Multiclass Perceptron

The perceptron presented above is a binary classifier, but we can extend it to account for multiple classes rather easily. The main difference is in how we set up weights and how we update said weights. For the binary case, we had one weight vector, which had a dimension equal to the number of features (plus the bias feature). For the multi-class case, we will have one weight vector for each class, so in the 3 class case, we have 3 weight vectors. In order to classify a sample, we compute a score for each class by taking the dot product of the feature vector with each of the weight vectors. Whichever class yields the highest score is the one we choose as our prediction.

For example, consider the 3-class case. Let our sample have features $\mathbf{f}(\mathbf{x}) = \begin{bmatrix} -2 & 3 & 1 \end{bmatrix}$ and our weights for classes 0, 1, and 2 be:

$$\mathbf{w}_0 = \begin{bmatrix} -2 & 2 & 1 \end{bmatrix}$$
$$\mathbf{w}_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix}$$
$$\mathbf{w}_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix}$$

Taking dot products for each class gives us scores $s_0 = 11, s_1 = 13, s_2 = 8$. We would thus predict that $\mathbf{x}$ belongs to class 1.

An important thing to note is that in actual implementation, we do not keep track of the weights as separate structures, we usually stack them on top of each other to create a weight matrix. This way, instead of doing as many dot products as there are classes, we can instead do a single matrix-vector multiplication. This tends to be much more efficient in practice (because matrix-vector multiplication usually has a highly optimized implementation).

In our above case, that would be:

$$\mathbf{W} = \begin{bmatrix} -2 & 2 & 1 \\ 0 & 3 & 4 \\ 1 & 4 & -2 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

And our label would be:

$$\arg\max(\mathbf{W}\mathbf{x}) = \arg\max(\begin{bmatrix} 11 \\ 13 \\ 8 \end{bmatrix}) = 1$$

Along with the structure of our weights, our weight update also changes when we move to a multi-class case. If we correctly classify our data point, then do nothing just like in the binary case. If we chose incorrectly, say we chose class $y \neq y^*$, then we add the feature vector to the weight vector for the true class to y^*, and subtract the feature vector from the weight vector corresponding to the predicted class y. In our above example, let's say that the correct class was class 2, but we predicted class 1. We would now take the weight vector corresponding to class 1 and subtract x from it,

$$\mathbf{w}_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \end{bmatrix}$$

Next we take the weight vector corresponding to the correct class, class 2 in our case, and add x to it:

$$\mathbf{w}_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix} + \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 7 & -1 \end{bmatrix}$$

What this amounts to is 'rewarding' the correct weight vector, 'punishing' the misleading, incorrect weight vector, and leaving alone an other weight vectors. With the difference in the weights and weight updates taken into account, the rest of the algorithm is essentially the same; cycle through every sample point, updating weights when a mistake is make, until you stop making mistakes.

In order to incorporate a bias term, do the same as we did for binary perceptron – add an extra feature of 1 to every feature vector, and an extra weight for this feature to every class's weight vector (this amounts to adding an extra column to the matrix form).

Summary

In this note, we introduced several fundamental principles of machine learning, including:

- Splitting our data into training data, validation data, and test data.
- The difference between supervised learning, which learns from labeled data, and unsupervised learning, which doesn't have labeled data and so attempts to infer inherent structure from it.

We then proceeded to discuss an number of supervised learning algorithms such as Naive Bayes, Linear Regression, and the Perceptron Algorithm.

- We covered the Naive Bayes algorithm and derived the maximum likelihood estimates of the unknown model parameters. We extended this idea to discuss the problem of overfitting in the context of Naive Bayes' and how this issue can be mitigated with Laplace smoothing.
- We talked about Linear Regression, a simple model where we predict real-valued outputs as linear combinations of our input features. We also derived the linear regression closed form solution using vector calculus.
- Finally, we talked about linear decision boundaries and the perceptron algorithm - a method for classification that repeatedly iterates over all our data and updates weight vectors when it classifies points incorrectly.