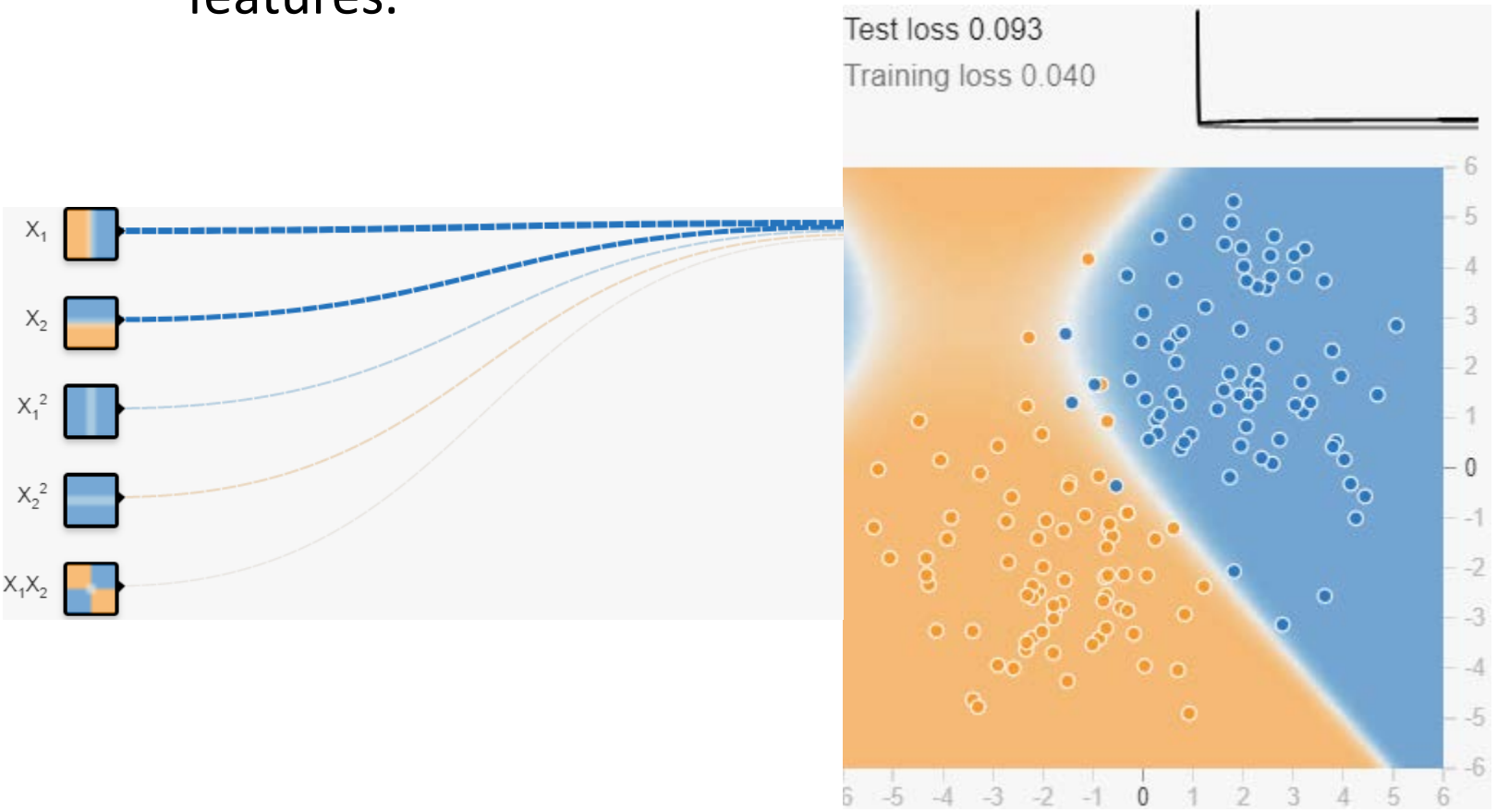
Machine Learning (Lecture 5)

UEM/IEM Summer 2018

Regularization for Simplicity

- Overcrossing?
  - Let's explore overuse of feature crosses.
  - Run the model as is, with all of the given cross-product features.

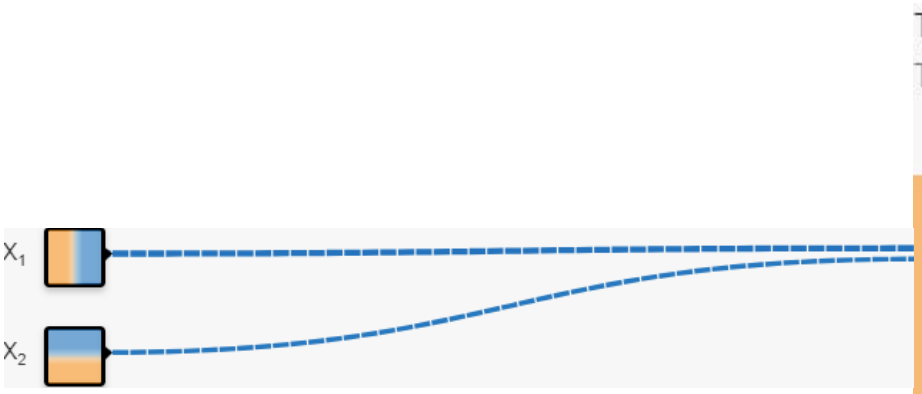


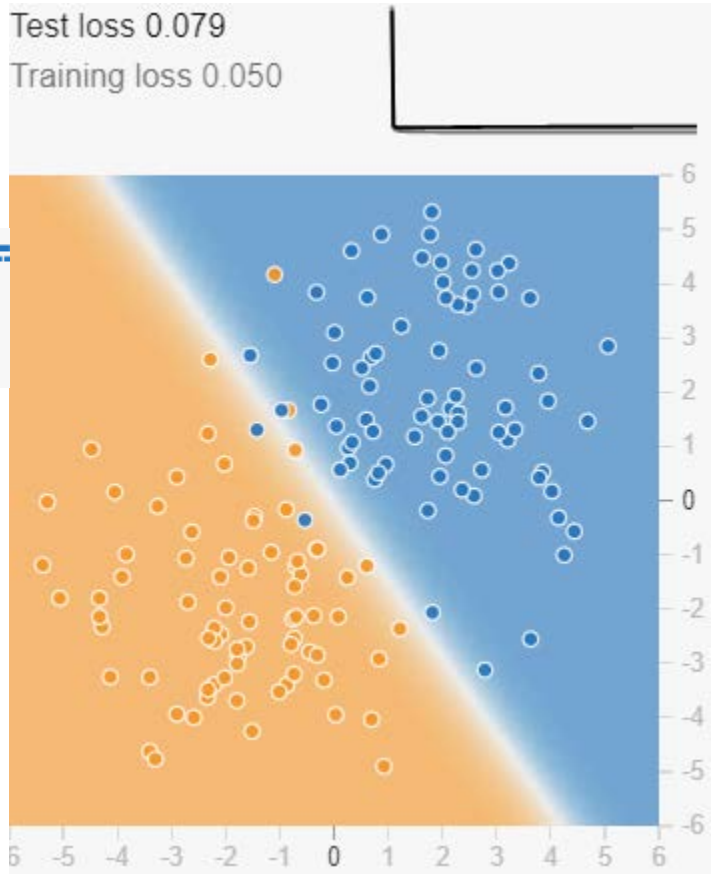
Regularization for Simplicity

- Surprisingly, the model's decision boundary looks kind of crazy. In particular, there's a region in the upper left that's hinting towards blue, even though there's no visible support for that in the data.
- Notice the relative thickness of the five lines running from INPUT to OUTPUT. These lines show the relative weights of the five features. The lines emanating from X_1 and X_2 are much thicker than those coming from the feature crosses. So, the feature crosses are contributing far less to the model than the normal (uncrossed) features.

Regularization for Simplicity

- Overcrossing?
  - Try removing various cross-product features to improve performance (albeit only slightly).





Regularization for Simplicity

- Removing all the feature crosses gives a saner model (there is no longer a curved boundary suggestive of overfitting) and makes the test loss converge.
- After 1,000 iterations, test loss should be a slightly lower value than when the feature crosses were in play (although your results may vary a bit, depending on the data set).
- The data in this exercise is basically linear data plus noise. If we use a model that is too complicated, such as one with too many crosses, we give it the opportunity to fit to the noise in the training data, often at the cost of making the model perform badly on test data.

Regularization for Simplicity

- **Regularization** means penalizing the complexity of a model to reduce overfitting.

Regularization for Simplicity: L_2 Regularization

- Consider the following **generalization curve**, which shows the loss for both the training set and validation set against the number of training iterations.

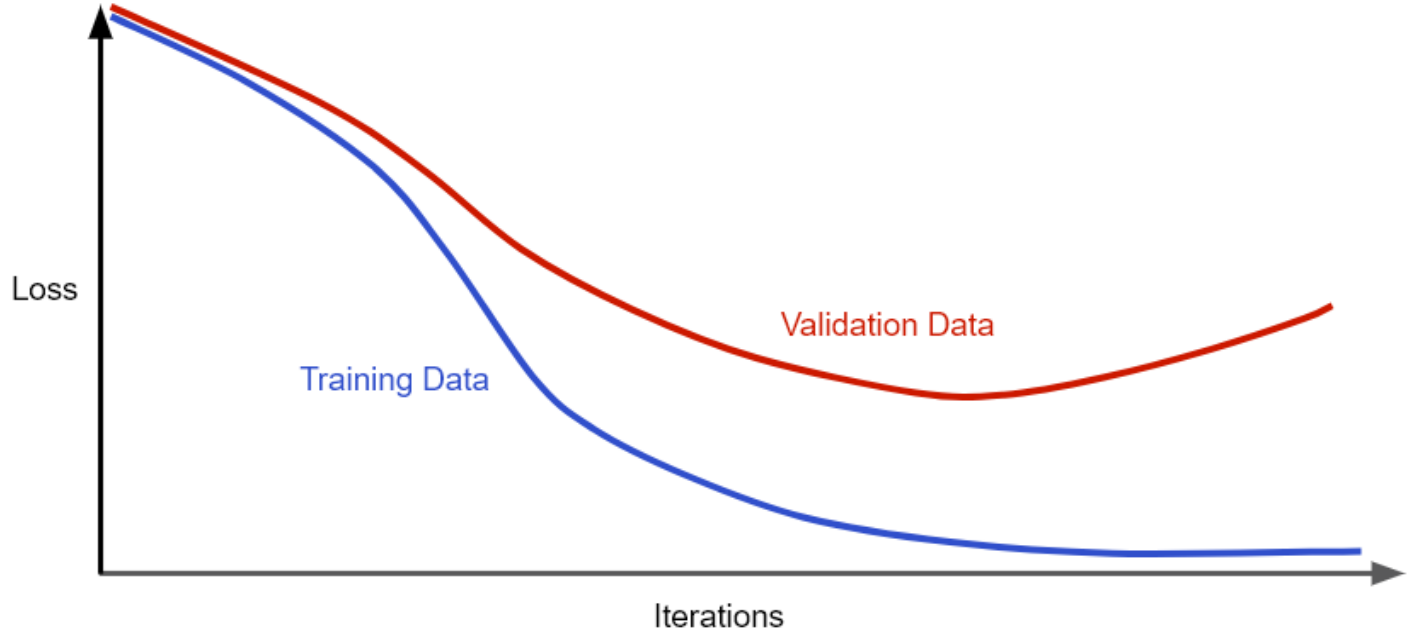


Figure 1. Loss on training set and validation set.

Regularization for Simplicity: L_2 Regularization

- The previous figure shows a model in which training loss gradually decreases, but validation loss eventually goes up.
- This generalization curve shows that the model is overfitting to the data in the training set.
- We could prevent overfitting by penalizing complex models, a principle called **regularization**.
- We will now minimize loss+complexity, which is called **structural risk minimization**:

  minimize(Loss(Data|Model) + complexity(Model))

Regularization for Simplicity: L_2 Regularization

- Our training optimization algorithm is now a function of two terms:
  - the **loss term**, which measures how well the model fits the data, and
  - the **regularization term**, which measures model complexity.
- Two common (and somewhat related) ways to think of model complexity:
  - Model complexity as a function of the *weights* of all the features in the model.
  - Model complexity as a function of the *total number of features* with nonzero weights.

Regularization for Simplicity: L_2 Regularization

- If model complexity is a function of weights, a feature weight with a high absolute value is more complex than a feature weight with a low absolute value.
- We can quantify complexity using the ***L_2* regularization** formula, which defines the regularization term as the sum of the squares of all the feature weights:

$$L_2 \text{ regularization term} = ||\boldsymbol{w}||_2^2 = w_1^2 + w_2^2 + \ldots + w_n^2$$

- In this formula, weights close to zero have little effect on model complexity, while outlier weights can have a huge impact.

Regularization for Simplicity: L_2 Regularization

- For example, a linear model with the following weights:

  $\{w_1 = 0.2, w_2 = 0.5, w_3 = 5, w_4 = 1, w_5 = 0.25, w_6 = 0.75\}$

  Has an L_2 regularization term of 26.915:

$$w_1^2 + w_2^2 + \mathbf{w_3^2} + w_4^2 + w_5^2 + w_6^2$$

$$= 0.2^2 + 0.5^2 + \mathbf{5^2} + 1^2 + 0.25^2 + 0.75^2$$

$$= 0.04 + 0.25 + \mathbf{25} + 1 + 0.0625 + 0.5625$$

$$= 26.915$$

- But w_3 (bolded above), with a squared value of 25, contributes nearly all the complexity. The sum of the squares of all five other weights adds just 1.915 to the L_2 regularization term.

Regularization for Simplicity: Lambda

- Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda**(also called the **regularization rate**). That is, model developers aim to do the following:

  minimize(Loss(Data|Model)+λ complexity(Model))

- Performing L_2 regularization has the following effect on a model
  - Encourages weight values toward 0 (but not exactly 0)
  - Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution.

Regularization for Simplicity: Lambda

- Increasing the lambda value strengthens the regularization effect. For example, the histogram of weights for a high value of lambda might look as shown in the following figure:

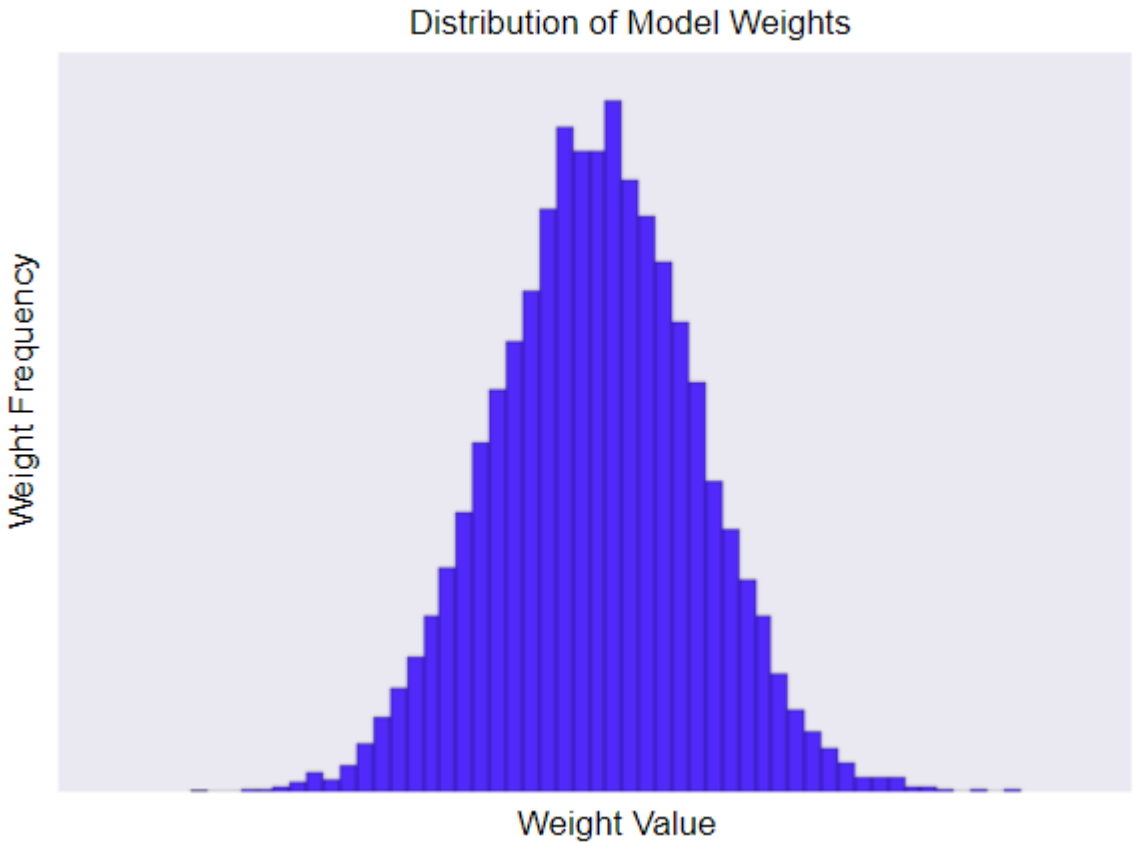


Figure 2. Histogram of weights.

Regularization for Simplicity: Lambda

- Lowering the value of lambda tends to yield a flatter histogram, as shown in the following figure:

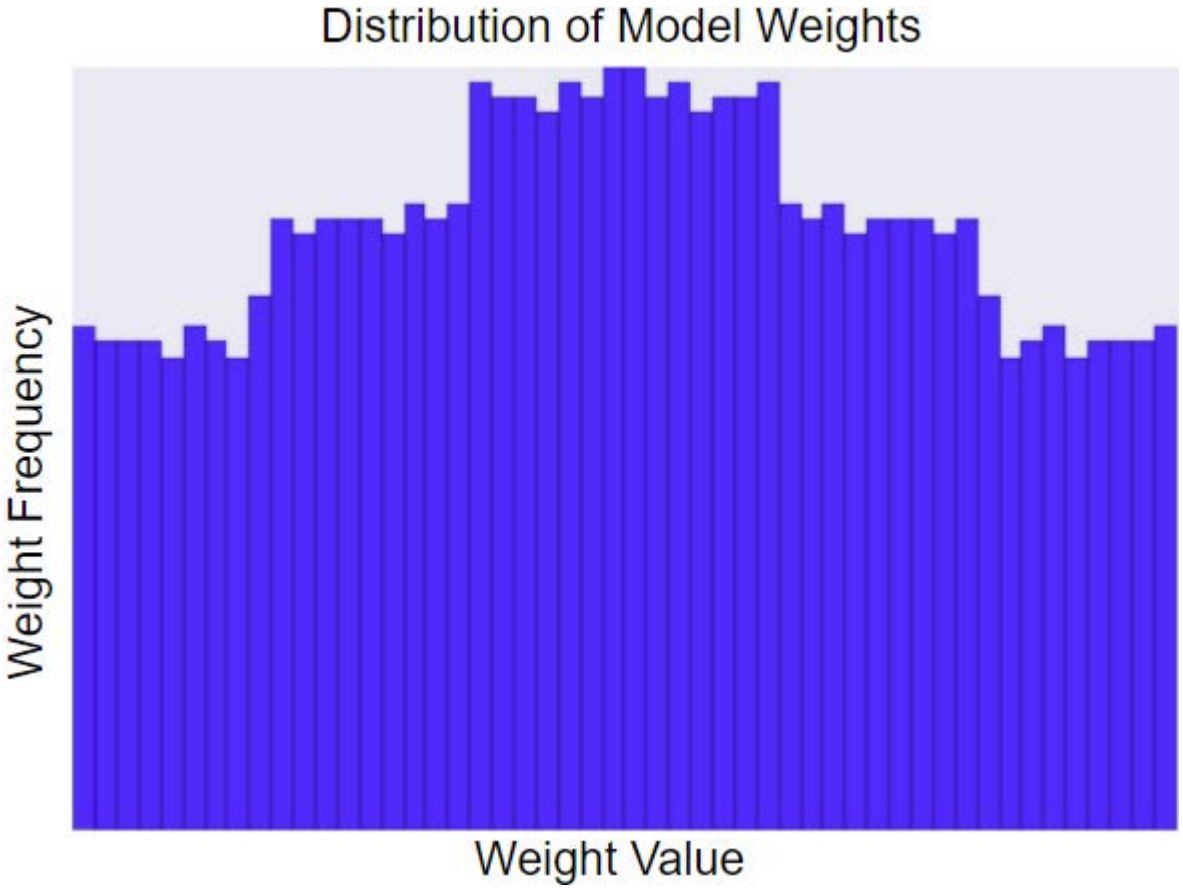
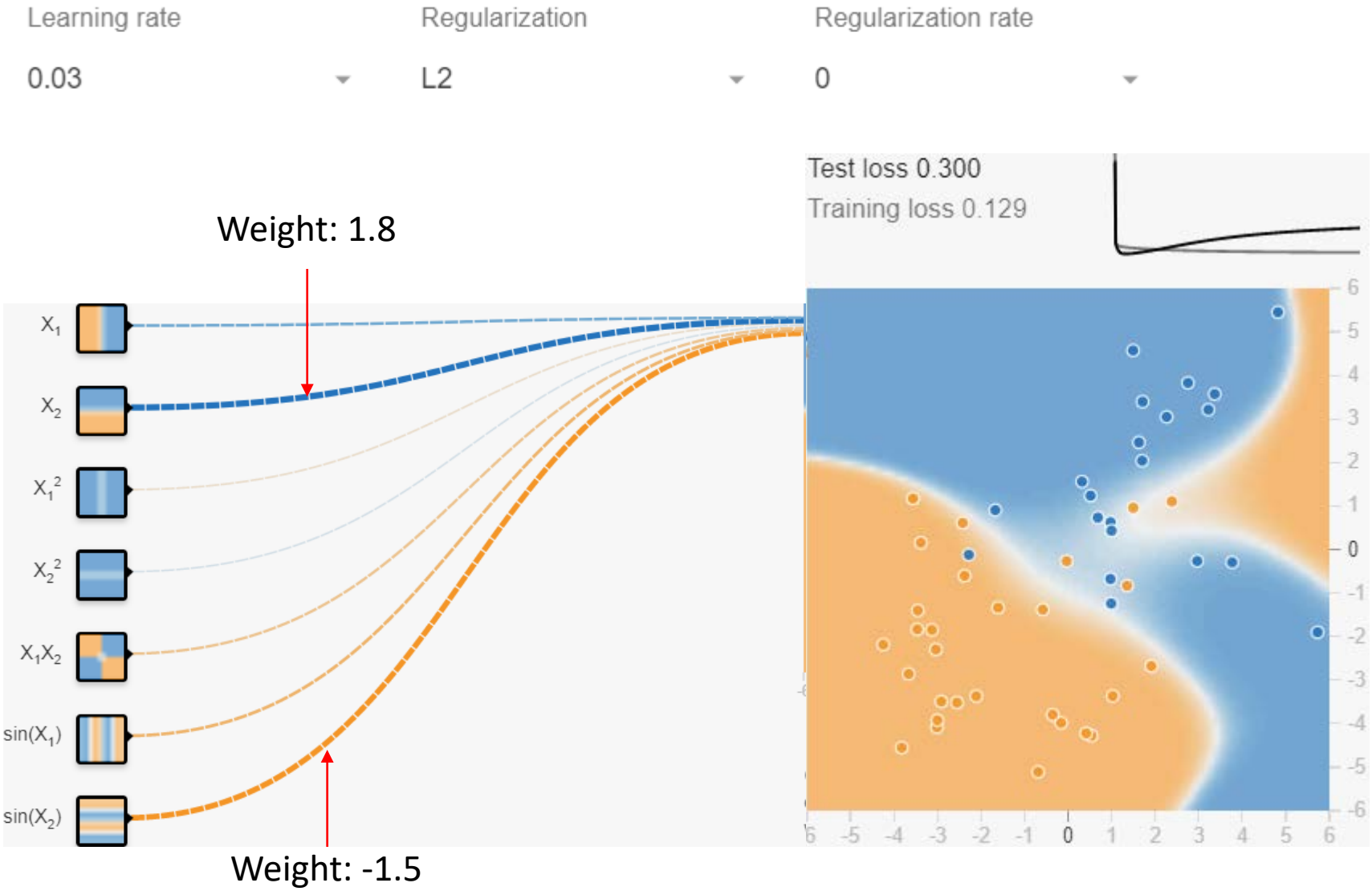


Figure 3. Histogram of weights produced by a lower lambda value.

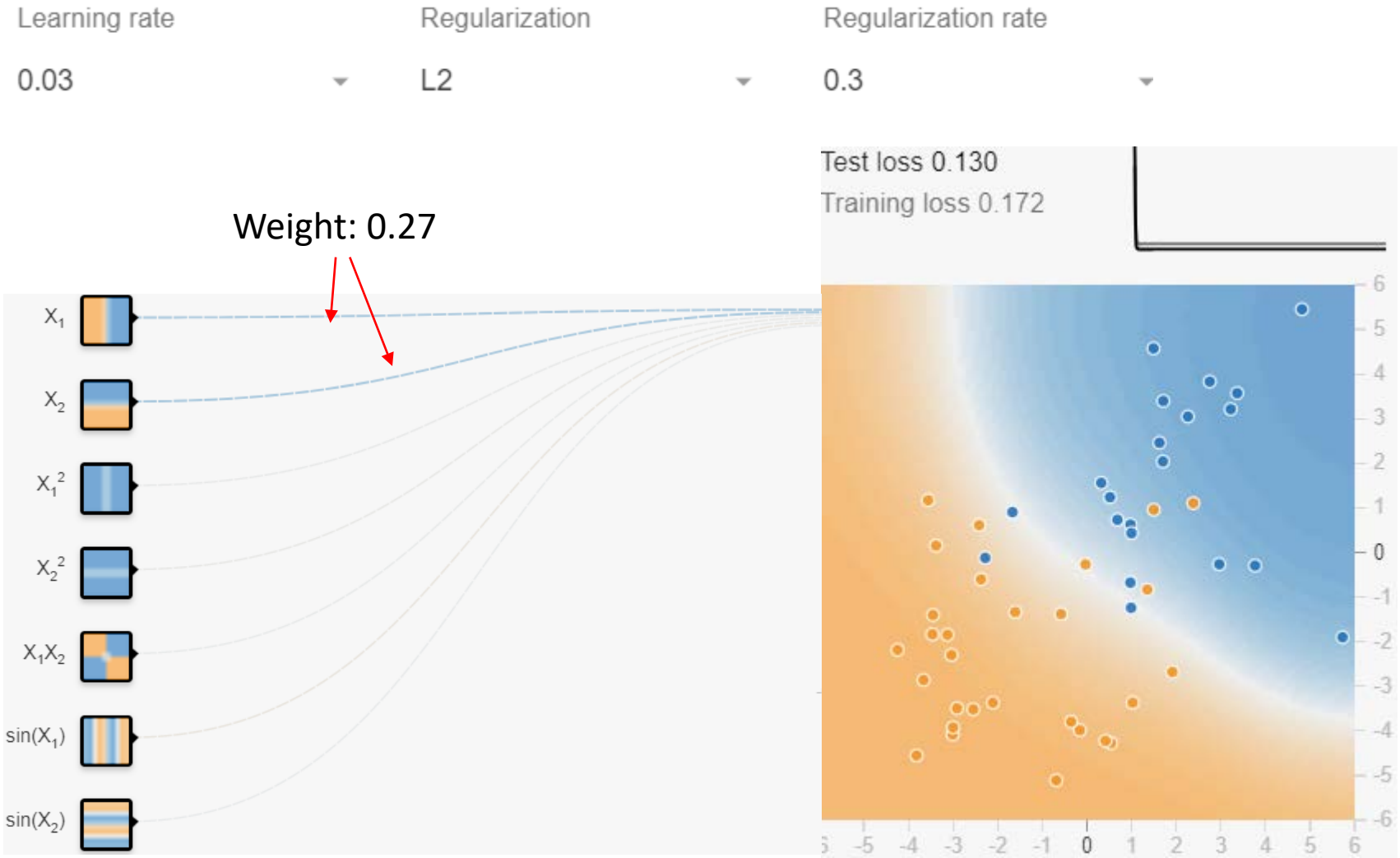
Regularization for Simplicity: Lambda

- When choosing a lambda value, the goal is to strike the right balance between simplicity and training-data fit:
  - If your lambda value is too high, your model will be simple, but you run the risk of *underfitting* your data. Your model won't learn enough about the training data to make useful predictions.
  - If your lambda value is too low, your model will be more complex, and you run the risk of *overfitting* your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data.
- The ideal value of lambda produces a model that generalizes well to new, previously unseen data. Unfortunately, that ideal value of lambda is data-dependent, so you'll need to do some tuning.

Regularization for Simplicity



Regularization for Simplicity



Regularization for Simplicity

- Increasing the regularization rate from 0 to 0.3 produces the following effects:
  - Test loss drops significantly.
  - The delta between Test loss and Training loss drops significantly.
  - The weights of the features and some of the feature crosses have lower absolute values, which implies that model complexity drops.

Regularization for Simplicity: Lambda

- Model developers tune the overall impact of the regularization term by multiplying its value by a scalar known as **lambda**(also called the **regularization rate**). That is, model developers aim to do the following:

  minimize(Loss(Data|Model)+λ complexity(Model))

- Performing L_2 regularization has the following effect on a model
  - Encourages weight values toward 0 (but not exactly 0)
  - Encourages the mean of the weights toward 0, with a normal (bell-shaped or Gaussian) distribution.

Quiz

- Imagine a linear model with 100 input features:
  - 10 are highly informative.
  - 90 are non-informative.
- Assume that all features have values between -1 and 1. Which of the following statements are true?

- L_2 regularization will encourage most of the non-informative weights to be exactly 0.0.
- L_2 regularization may cause the model to learn a moderate weight for some **non-informative** features.
- L_2 regularization will encourage many of the non-informative weights to be nearly (but not exactly) 0.0.

Quiz

- Imagine a linear model with 100 input features:
  - 10 are highly informative.
  - 90 are non-informative.
- Assume that all features have values between -1 and 1. Which of the following statements are true?

- L_2 regularization will encourage most of the non-informative weights to be exactly 0.0.
- L_2 regularization may cause the model to learn a moderate weight for some **non-informative** features.
- L_2 regularization will encourage many of the non-informative weights to be nearly (but not exactly) 0.0.

Quiz

- Imagine a linear model with two strongly correlated features; that is, these two features are nearly identical copies of one another but one feature contains a small amount of random noise. If we train this model with L_2regularization, what will happen to the weights for these two features?

- Both features will have roughly equal, moderate weights.
- One feature will have a large weight; the other will have a weight of **exactly** 0.0.
- One feature will have a large weight; the other will have a weight of **almost** 0.0.

Quiz

- Imagine a linear model with two strongly correlated features; that is, these two features are nearly identical copies of one another but one feature contains a small amount of random noise. If we train this model with L_2regularization, what will happen to the weights for these two features?

- Both features will have roughly equal, moderate weights.
- One feature will have a large weight; the other will have a weight of **exactly** 0.0.
- One feature will have a large weight; the other will have a weight of **almost** 0.0.

Logistic Regression

- Instead of predicting *exactly* 0 or 1, **logistic regression** generates a probability—a value *between* 0 and 1, exclusive.
- For example, consider a logistic regression model for spam detection.
- If the model infers a value of 0.932 on a particular email message, it implies a 93.2% probability that the email message is spam.
- More precisely, it means that in the limit of *infinite* training examples, the set of examples for which the model predicts 0.932 will actually be spam 93.2% of the time and the remaining 6.8% will not.

Logistic Regression: Calculating a Probability

- Many problems require a probability estimate as output.
- Logistic regression is an extremely efficient mechanism for calculating probabilities.
- Practically speaking, you can use the returned probability in either of the following two ways:
  - "As is"
  - Converted to a binary category.

Logistic Regression: Calculating a Probability

- Let's consider how we might use the probability "as is." Suppose we create a logistic regression model to predict the probability that a dog will bark during the middle of the night. We'll call that probability:

  p(bark | night)

- If the logistic regression model predicts a p(bark | night) of 0.05, then over a year, the dog's owners should be startled awake approximately 18 times:

  startled = p(bark | night) * nights

  18 ~= 0.05 * 365

Logistic Regression: Calculating a Probability

- In many cases, you'll map the logistic regression output into the solution to a binary classification problem, in which the goal is to correctly predict one of two possible labels (e.g., "spam" or "not spam").
- ou might be wondering how a logistic regression model can ensure output that always falls between 0 and 1.
- As it happens, a **sigmoid function**, defined as follows, produces output having those same characteristics:

$$y = \frac{1}{1 + e^{-z}}$$

Logistic Regression: Calculating a Probability

- The sigmoid function yields the following plot:

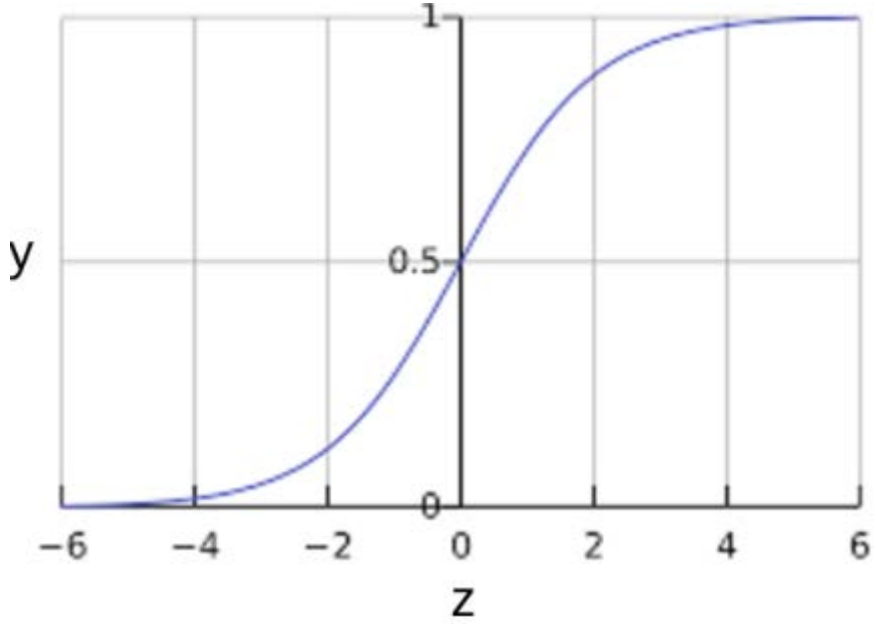


Figure 4: Sigmoid function.

Logistic Regression: Calculating a Probability

- If z represents the output of the linear layer of a model trained with logistic regression, then sigmoid(z) will yield a value (a probability) between 0 and 1. In mathematical terms:

$$y' = \frac{1}{1 + e^{-(z)}}$$

- where:
  - y' is the output of the logistic regression model for a particular example.
  - z is $b + w_1x_1 + w_2x_2 + \ldots w_Nx_N$
    - The w values are the model's learned weights and bias.
    - The x values are the feature values for a particular example.

Logistic Regression: Calculating a Probability

- Note that z is also referred to as the log-odds because the inverse of the sigmoid states that z can be defined as the log of the probability of the "1" label (e.g., "dog barks") divided by the probability of the "0" label (e.g., "dog doesn't bark"):

$$z = log(\frac{y}{1-y})$$

Logistic Regression: Calculating a Probability

- Here is the sigmoid function with ML labels:

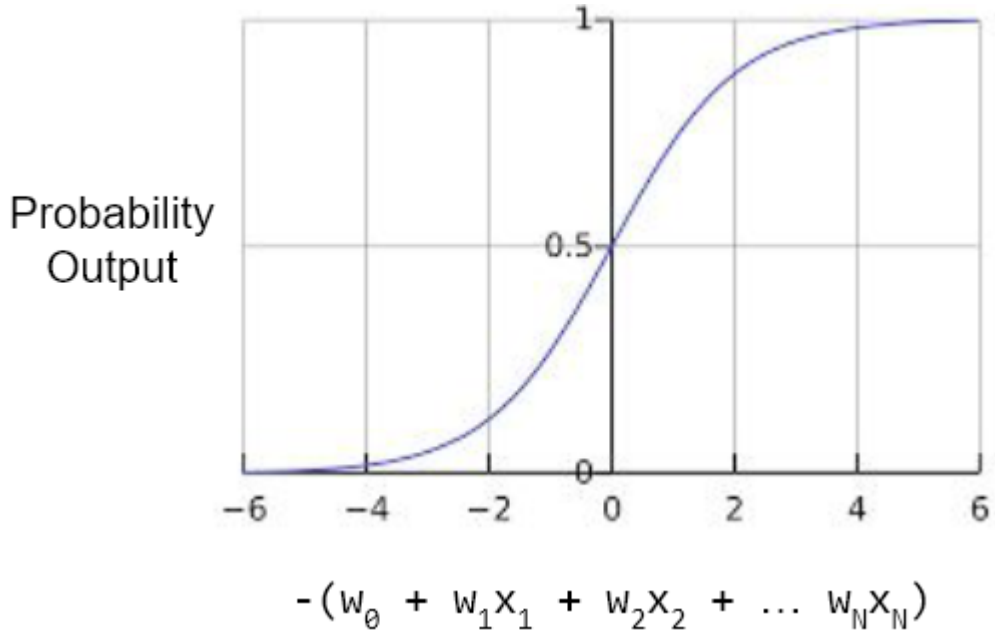


Figure 5: Logistic regression output.

Logistic Regression: Calculating a Probability

- A sample logistic regression inference calculation
  - Suppose we had a logistic regression model with three features that learned the following bias and weights:
    - b = 1, w_1 = 2 w_2 = -1, w_3 = 5
  - Further suppose the following feature values for a given example:
    - x_1 = 0, x_2 = 10, x_3 = 2
  - Therefore, the log-odds:

$$b + w_1 x_1 + w_2 x_2 + w_3 x_3$$

  - will be: (1) + (2)(0) + (-1)(10) + (5)(2) = 1
  - Consequently, the logistic regression prediction for this particular example will be 0.731:

$$y' = \frac{1}{1 + e^{-(1)}} = 0.731$$

Logistic Regression: Calculating a Probability

- A sample logistic regression inference calculation (cont'd)

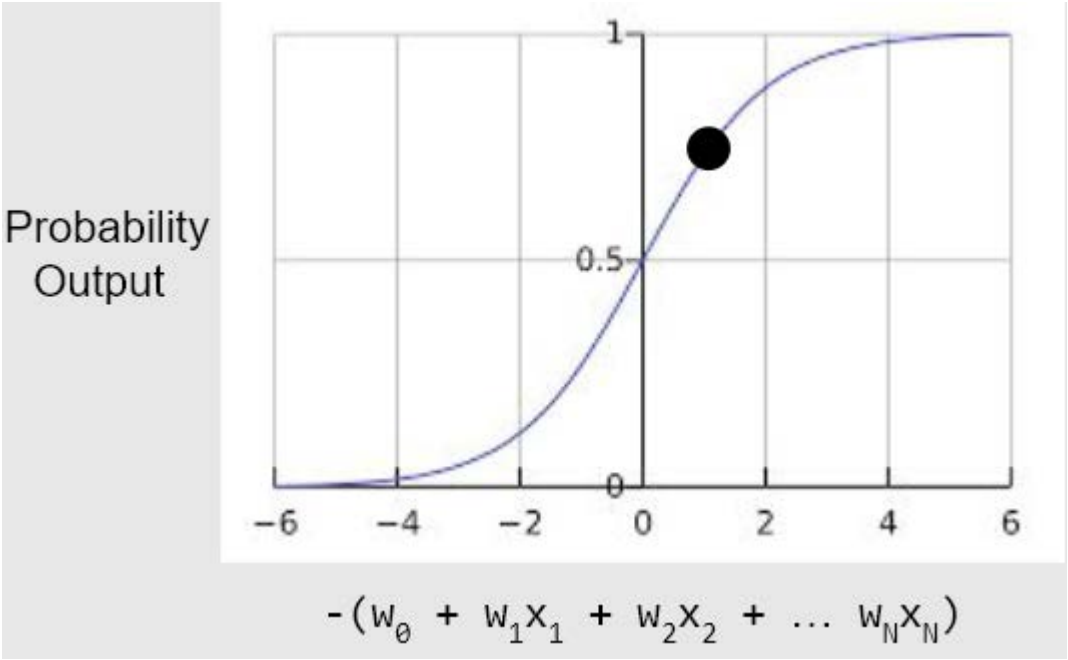


Figure 6: 73.1% probability.

Reference

- This lecture note has been developed based on the machine learning crash course at Google, which is under *Creative Commons Attribution 3.0 License*.