Machine Learning (Lecture 3)

UEM/IEM Summer 2018

Generalization: Peril of Overfitting

- **Generalization** refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model.
- In order to develop some intuition about the concept of generalization, you're going to look at three figures.
- Assume that each dot in these figures represents a tree's position in a forest.
- The two colors have the following meanings:
  - The blue dots represent sick trees.
  - The orange dots represent healthy trees.

Generalization: Peril of Overfitting

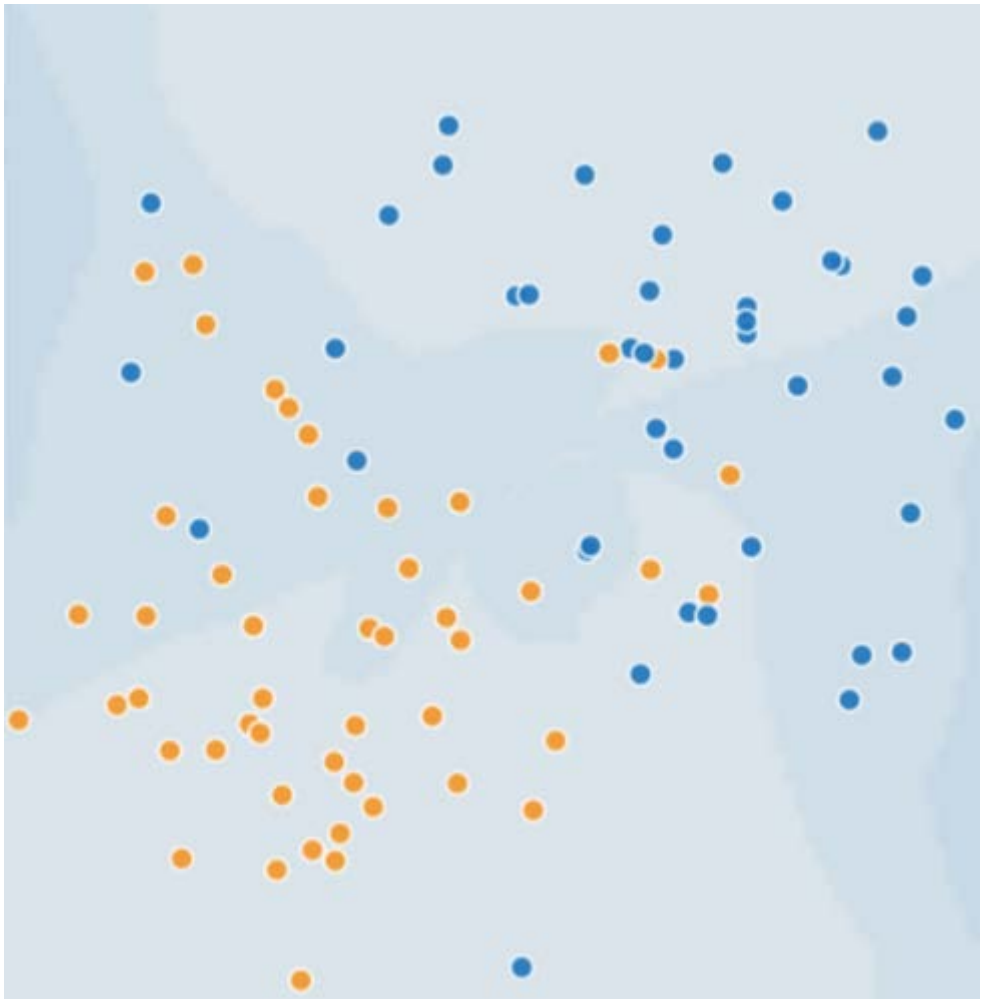


Figure 1. Sick (blue) and healthy (orange) trees.

Generalization: Peril of Overfitting

- Can you imagine a good model for predicting subsequent sick or healthy trees? Take a moment to mentally draw an arc that divides the blues from the oranges, or mentally lasso a batch of oranges or blues.

Generalization: Peril of Overfitting

- A machine learning model separating the sick trees from the healthy trees. Note that this model produced a very low loss.

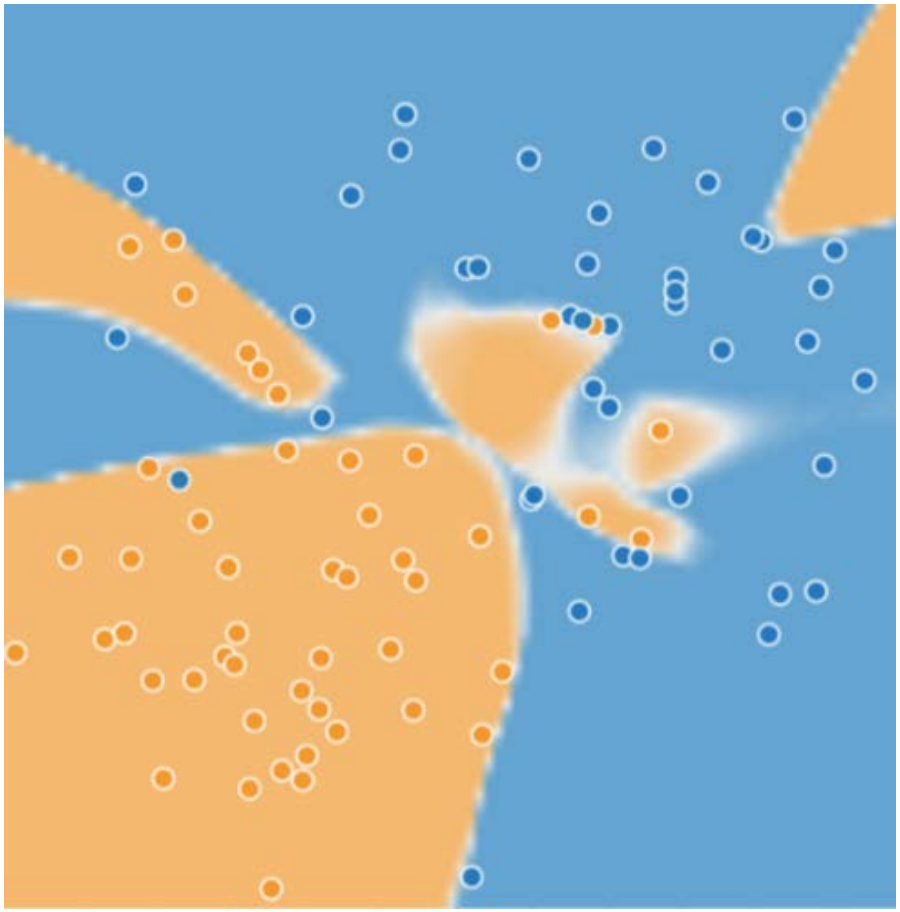


Figure 2. A complex model for distinguishing sick from healthy trees.

Generalization: Peril of Overfitting

- Figure 3 shows what happened when we added new data to the model. It turned out that the model adapted very poorly to the new data. Notice that the model miscategorized much of the new data.

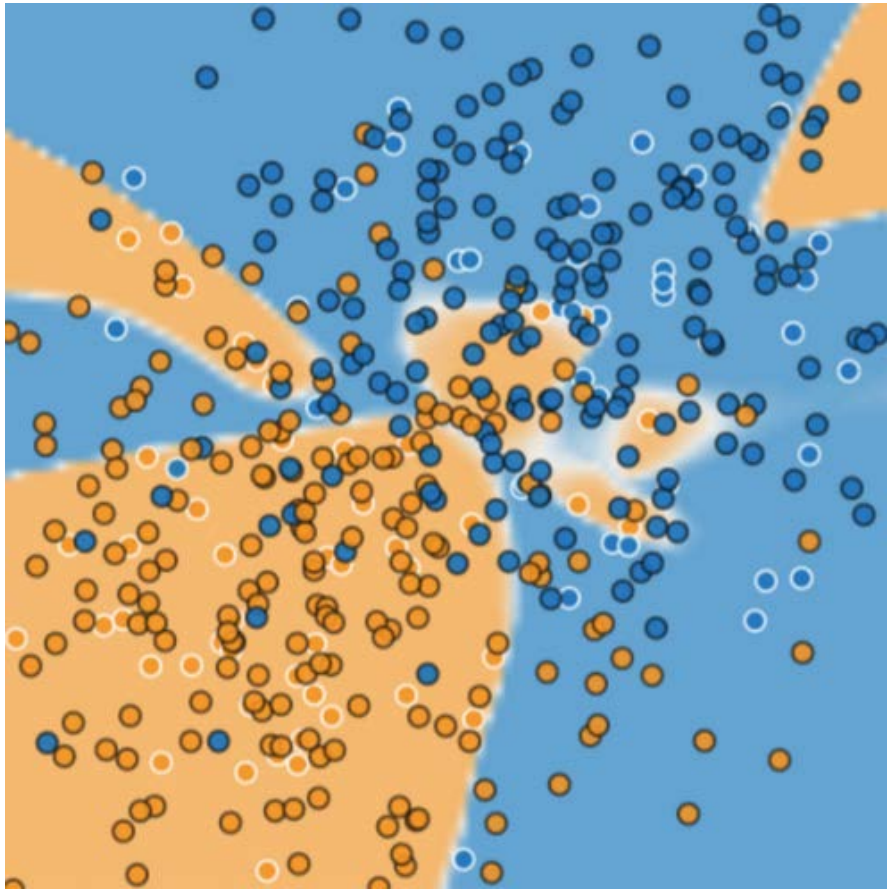


Figure 3. The model did a bad job predicting new data.

Generalization: Peril of Overfitting

- The model shown in Figures 2 and 3 **overfits** the peculiarities of the data it trained on.
- An overfit model gets a low loss during training but does a poor job predicting new data.
- If a model fits the current sample well, how can we trust that it will make good predictions on new data?
- Overfitting is caused by making a model more complex than necessary.
- The fundamental tension of machine learning is between fitting our data well, but also fitting the data as simply as possible.

Generalization: Peril of Overfitting

- Machine learning's goal is to predict well on new data drawn from a (hidden) true probability distribution.
- Unfortunately, the model can't see the whole truth; the model can only sample from a training data set.
- If a model fits the current examples well, how can you trust the model will also make good predictions on never-before-seen examples?
- William of Ockham, a 14th century friar and philosopher, loved simplicity. He believed that scientists should prefer simpler formulas or theories over more complex ones. To put Ockham's razor in machine learning terms:

  The less complex an ML model, the more likely that a good empirical result is not just due to the peculiarities of the sample.

Generalization: Peril of Overfitting

- In modern times, we've formalized Ockham's razor into the fields of **statistical learning theory** and **computational learning theory**.
- These fields have developed **generalization bounds**--a statistical description of a model's ability to generalize to new data based on factors such as:
  - the complexity of the model
  - the model's performance on training data
- While the theoretical analysis provides formal guarantees under idealized assumptions, they can be difficult to apply in practice.

Generalization: Peril of Overfitting

- A machine learning model aims to make good predictions on new, previously unseen data.
- But if you are building a model from your data set, how would you get the previously unseen data?
- One way is to divide your data set into two subsets:
  - **training set**—a subset to train a model.
  - **test set**—a subset to test the model.
- Good performance on the test set is a useful indicator of good performance on the new data in general, assuming that:
  - The test set is large enough.
  - You don't cheat by using the same test set over and over.

Generalization: Peril of Overfitting

- The following three basic assumptions guide generalization:
  - We draw examples **independently and identically** (**i.i.d**) at random from the distribution. In other words, examples don't influence each other. (An alternate explanation: i.i.d. is a way of referring to the randomness of variables.)
  - The distribution is **stationary**; that is the distribution doesn't change within the data set.
  - We draw examples from partitions from the **same distribution.**

Generalization: Peril of Overfitting

- In practice, we sometimes violate these assumptions. For example:
  - Consider a model that chooses ads to display. The i.i.d. assumption would be violated if the model bases its choice of ads, in part, on what ads the user has previously seen.
  - Consider a data set that contains retail sales information for a year. User's purchases change seasonally, which would violate stationarity.
  - When we know that any of the preceding three basic assumptions are violated, we must pay careful attention to metrics.

Training and Test Sets: Splitting Data

- We've learned the idea of dividing your data set into two subsets:
  - **training set**—a subset to train a model.
  - **test set**—a subset to test the trained model.
- You could imagine slicing the single data set as follows:

Figure 4. Slicing a single data set into a training set and test set.

Training and Test Sets: Splitting Data

- Make sure that your test set meets the following two conditions:
  - Is large enough to yield statistically meaningful results.
  - Is representative of the data set as a whole. In other words, don't pick a test set with different characteristics than the training set.
- Assuming that your test set meets the preceding two conditions, your goal is to create a model that generalizes well to new data.
- Our test set serves as a proxy for new data. For example, consider the following figure.

Training and Test Sets: Splitting Data

- Consider the following figure.

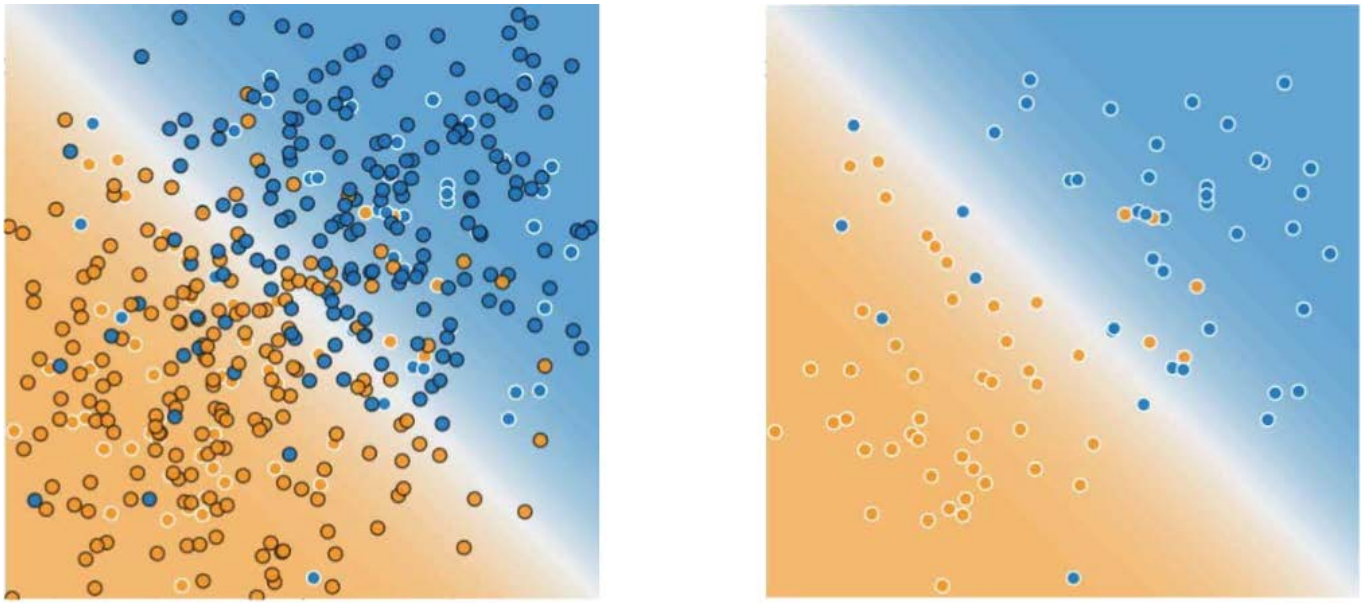


Figure 5. Validating the trained model against test data.

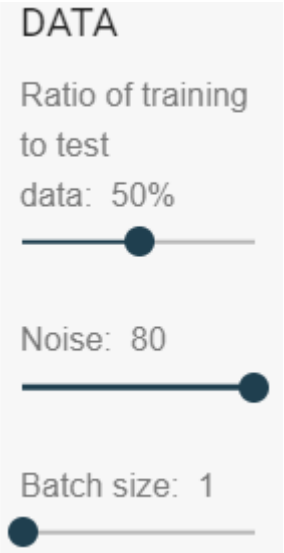
- Notice that the model learned for the training data is very simple.
- This model doesn't do a perfect job—a few predictions are wrong.
- However, this model does about as well on the test data as it does on the training data. In other words, this simple model does not overfit the training data.

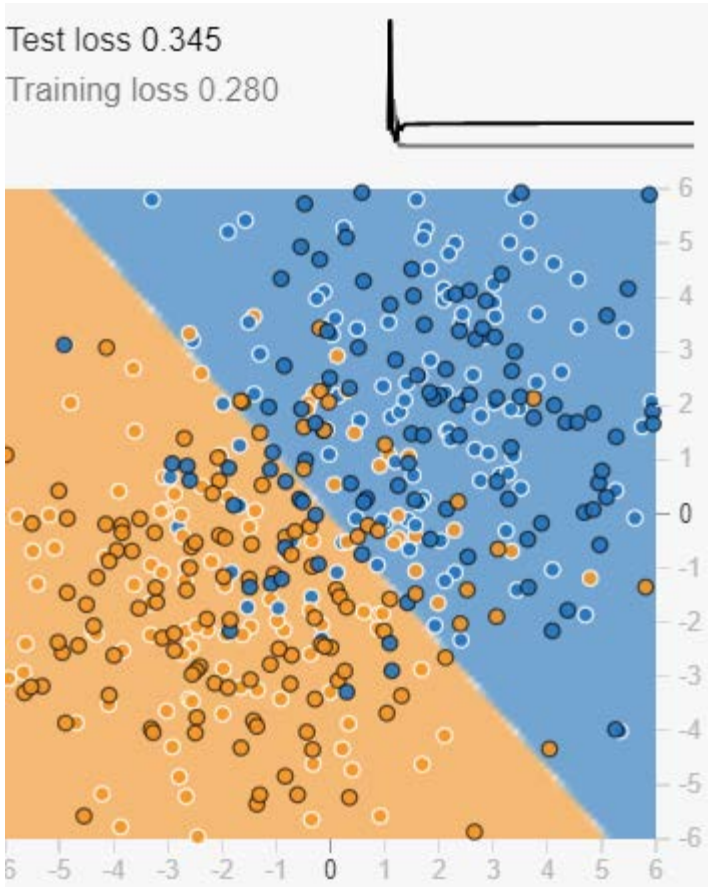
Training and Test Sets: Splitting Data

- **Never train on test data.**
- If you are seeing surprisingly good results on your evaluation metrics, it might be a sign that you are accidentally training on the test set.
- For example, high accuracy might indicate that test data has leaked into the training set.

Training and Test Sets

- Experiments
  - The training examples have a white outline.
  - The test examples have a black outline.
- Experiment #1

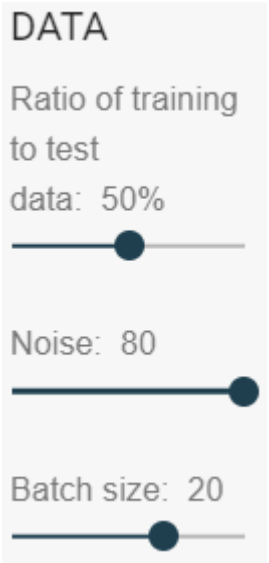


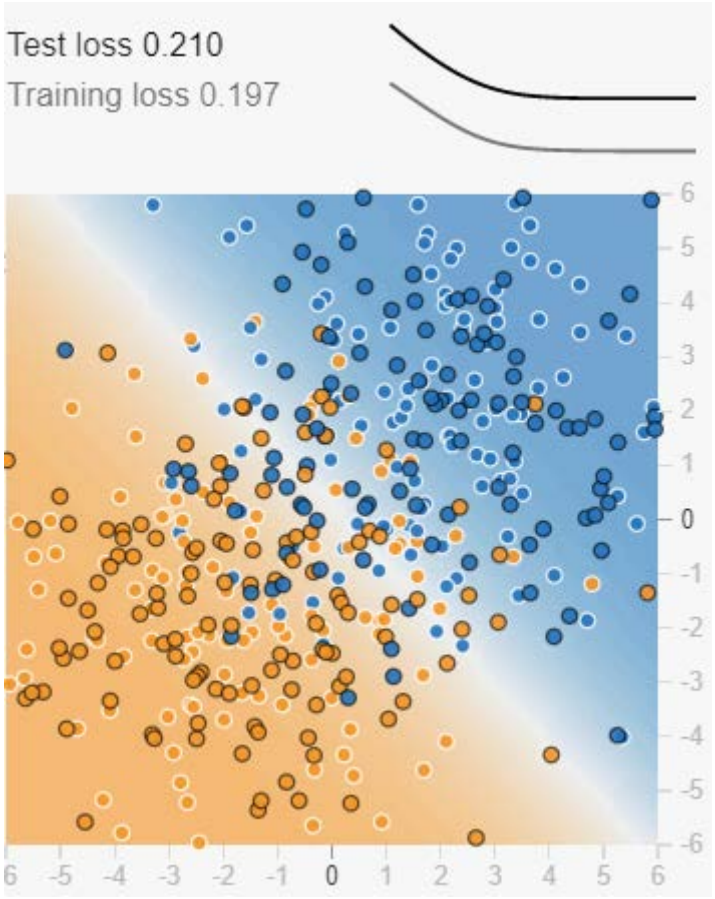


- With learning rate set to 3 (the initial setting), Test loss is significantly higher than Training loss.

Training and Test Sets

- Experiment #2

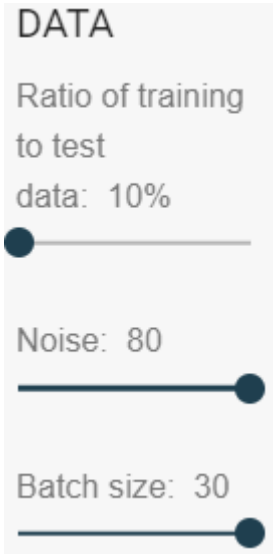


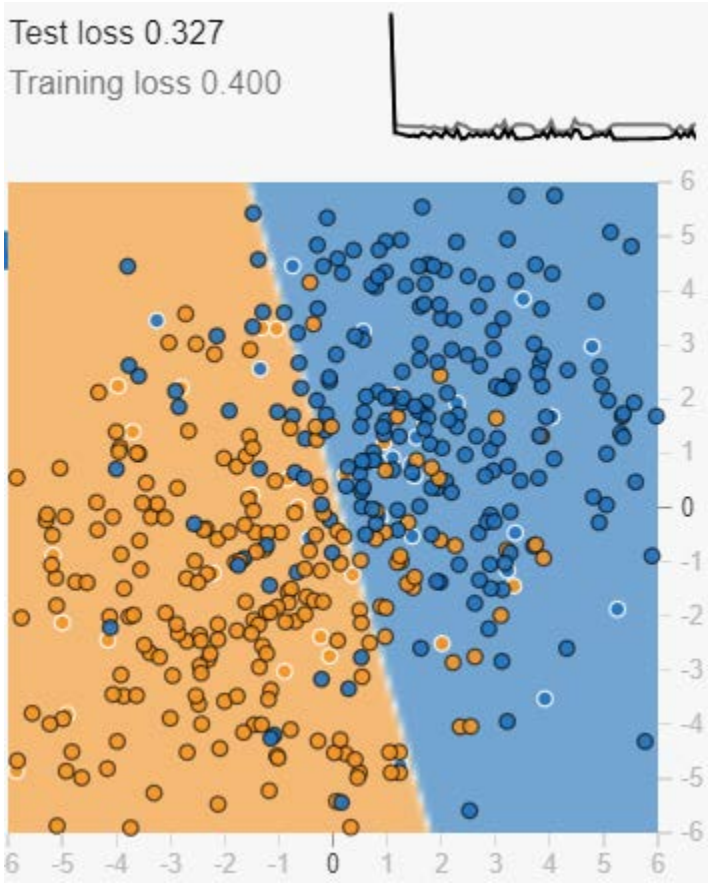


- By reducing learning rate (for example, to 0.001), Test loss drops to a value much closer to Training loss. In most runs, increasing Batch size does not influence Training loss or Test loss significantly. However, in a small percentage of runs, increasing Batch size to 20 or greater causes Test loss to drop slightly below Training loss.

Training and Test Sets

- Experiment #3





- Reducing the ratio of training to test data from 50% to 10% dramatically lowers the number of data points in the training set. With so little data, high batch size and high learning rate cause the training model to jump around chaotically (jumping repeatedly over the minimum point).

Validation: Check Your Intuition

- In doing a process of using a test set and a training set to drive iterations of model development, we train on the training data and evaluate on the test data on each iteration, using the evaluation results on test data to guide choices of and changes to various model hyperparameters like learning rate and features.
- However, the more often we evaluate on a given test set, the more we are at risk for implicitly overfitting to that one test set.

Validation

- Partitioning a data set into a training set and test set lets you judge whether a given model will generalize well to new data.
- However, using only two partitions may be insufficient when doing many rounds of hyperparameter tuning.

Validation: Another Partition

- Partitioning a data set into a training set and a test set enabled you to train on one set of examples and then to test the model against a different set of examples. With two partitions, the workflow could look as follows:

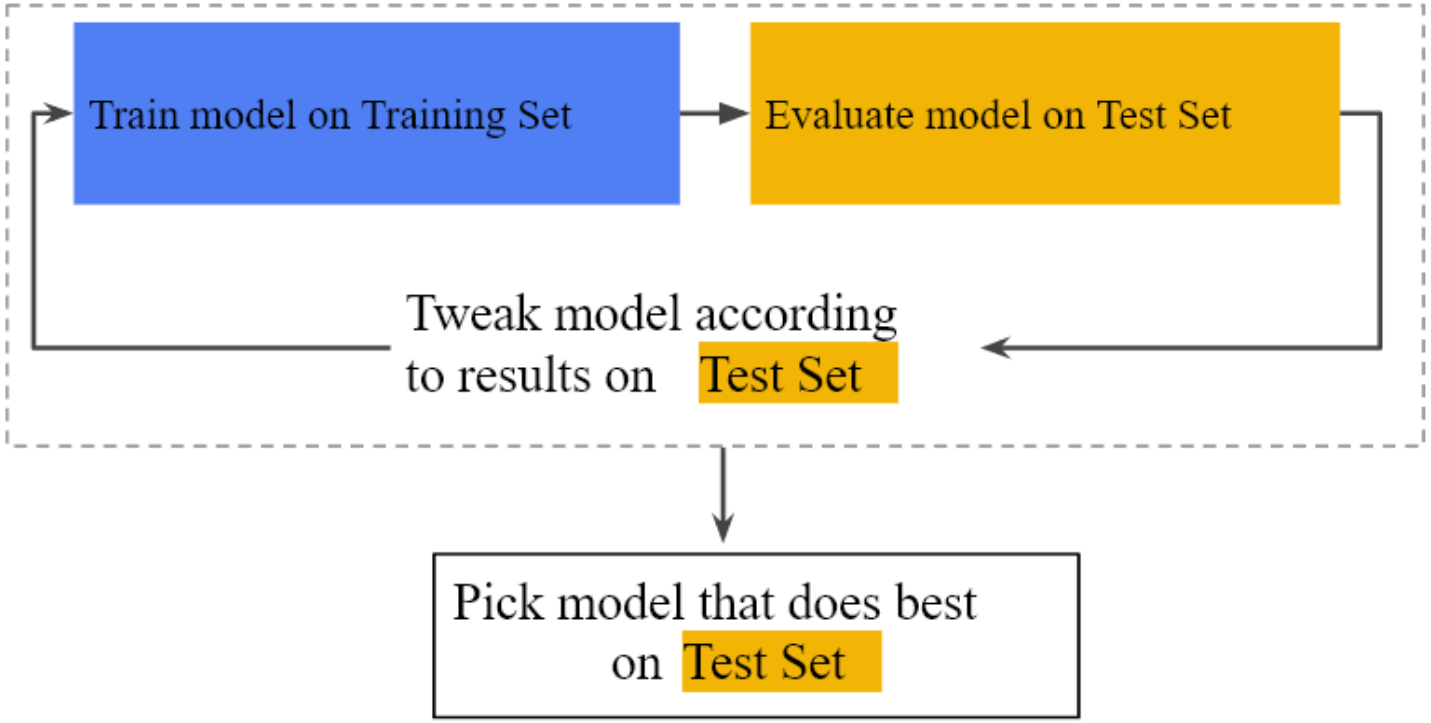


Figure 5. A possible workflow?

Validation: Another Partition

- "Tweak model" means adjusting anything about the model you can dream up—from changing the learning rate, to adding or removing features, to designing a completely new model from scratch. At the end of this workflow, you pick the model that does best on the *test set*.
- Dividing the data set into two sets is a good idea, but not a panacea. You can greatly reduce your chances of overfitting by partitioning the data set into the three subsets shown in the following figure:

Figure 6. Slicing a single data set into three subsets.

Validation: Another Partition

- Use the **validation set** to evaluate results from the training set. Then, use the test set to double-check your evaluation *after* the model has "passed" the validation set. The following figure shows this new workflow:

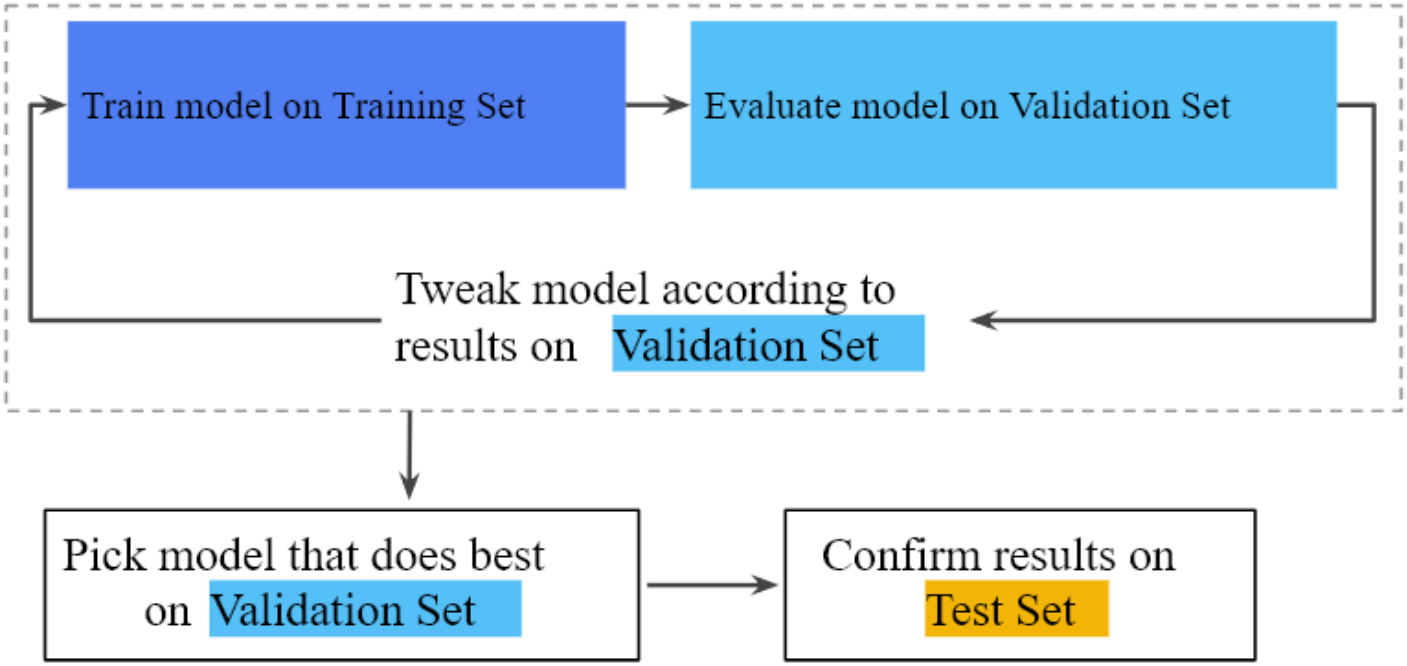


Figure 7. A better workflow.

Validation: Another Partition

- In this improved workflow:
  1. Pick the model that does best on the validation set.
  2. Double-check that model against the test set.
- This is a better workflow because it creates fewer exposures to the test set.
- Test sets and validation sets "wear out" with repeated use. That is, the more you use the same data to make decisions about hyperparameter settings or other model improvements, the less confidence you'll have that these results actually generalize to new, unseen data.
- Note that validation sets typically wear out more slowly than test sets.
- If possible, it's a good idea to collect more data to "refresh" the test set and validation set. Starting anew is a great reset.

Representation

- A machine learning model can't directly see, hear, or sense input examples. Instead, you must create a **representation** of the data to provide the model with a useful vantage point into the data's key qualities. That is, in order to train a model, you must choose the set of features that best represent the data.

Representation: Feature Engineering

- In traditional programming, the focus is on code. In machine learning projects, the focus shifts to representation.
- That is, one way developers hone a model is by adding and improving its features.

Representation: Feature Engineering

- Mapping Raw Data to Features

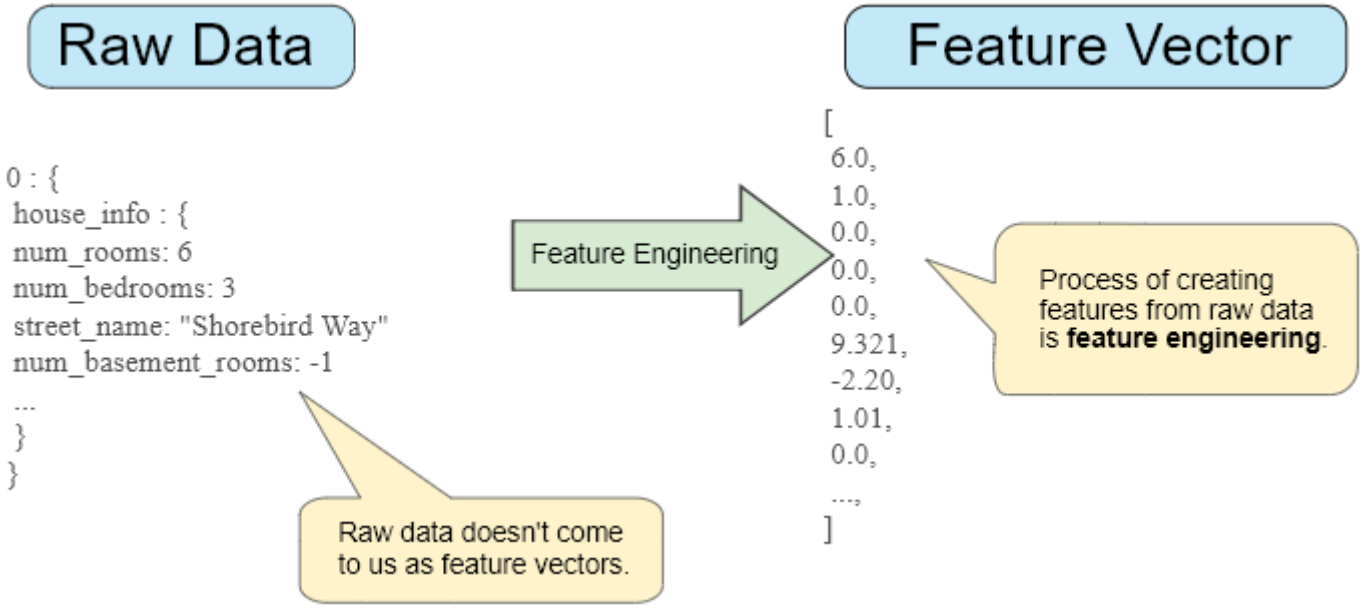


Figure 8. Feature engineering maps raw data to ML features.

- The left side of Figure 8 illustrates raw data from an input data source; the right side illustrates a **feature vector**, which is the set of floating-point values comprising the examples in your data set.
- **Feature engineering** means transforming raw data into a feature vector. Expect to spend significant time doing feature engineering.
- Many machine learning models must represent the features as real-numbered vectors since the feature values must be multiplied by the model weights.

Representation: Feature Engineering

- Mapping numeric values

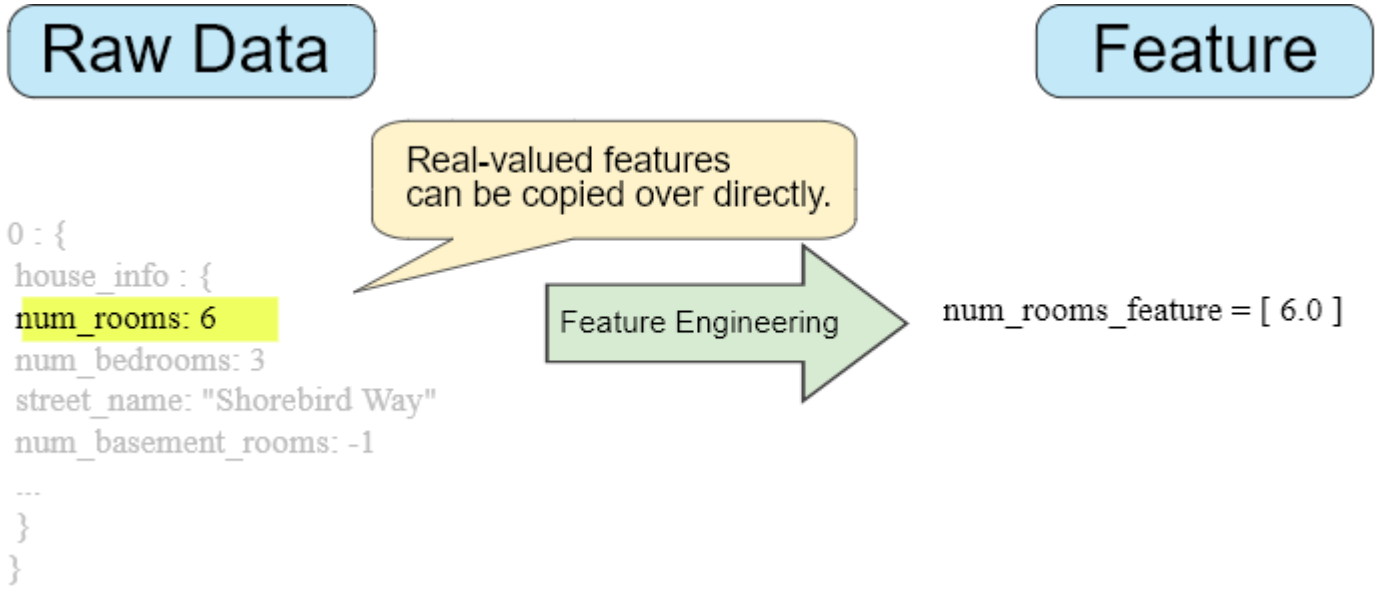


Figure 9. Mapping integer values to floating-point values.

- Integer and floating-point data don't need a special encoding because they can be multiplied by a numeric weight. As suggested in Figure 9, converting the raw integer value 6 to the feature value 6.0 is trivial.

Representation: Feature Engineering

- Mapping categorical values
  - Categorical features have a discrete set of possible values. For example, there might be a feature called street_name with options that include:

    {'Charleston Road', 'North Shoreline Boulevard’, 'Shorebird Way', 'Rengstorff Avenue’}
  - Since models cannot multiply strings by the learned weights, we use feature engineering to convert strings to numeric values.
  - We can accomplish this by defining a mapping from the feature values, which we'll refer to as the **vocabulary** of possible values, to integers. Since not every street in the world will appear in our dataset, we can group all other streets into a catch-all "other" category, known as an **OOV (out-of-vocabulary) bucket**.

Representation: Feature Engineering

- Using this approach, here's how we can map our street names to numbers:
  - map Charleston Road to 0
  - map North Shoreline Boulevard to 1
  - map Shorebird Way to 2
  - map Rengstorff Avenue to 3
  - map everything else (OOV) to 4
- However, if we incorporate these index numbers directly into our model, it will impose some constraints that might be problematic.

Representation: Feature Engineering

- To overcome the possible problems, we can instead create a binary vector for each categorical feature in our model that represents values as follows:
  - For values that apply to the example, set corresponding vector elements to 1.
  - Set all other elements to 0.
- The length of this vector is equal to the number of elements in the vocabulary. This representation is called a **one-hot encoding** when a single value is 1, and a **multi-hot encoding** when multiple values are 1.

Representation: Feature Engineering

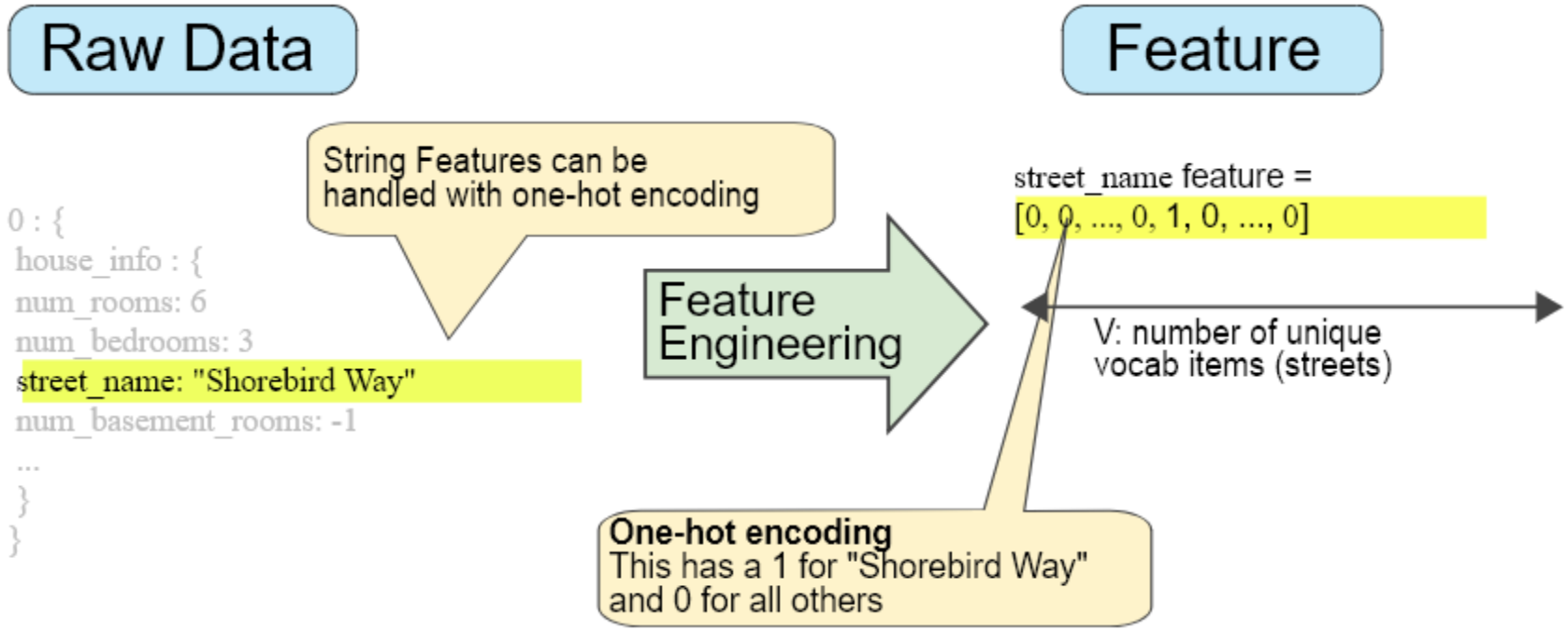


Figure 10. Mapping street address via one-hot encoding.

- Figure 10 illustrates a one-hot encoding of a particular street: Shorebird Way. The element in the binary vector for Shorebird Way has a value of 1, while the elements for all other streets have values of 0.
- This approach effectively creates a Boolean variable for every feature value (e.g., street name). Here, if a house is on Shorebird Way then the binary value is 1 only for Shorebird Way. Thus, the model uses only the weight for Shorebird Way.
- Similarly, if a house is at the corner of two streets, then two binary values are set to 1, and the model uses both their respective weights.

Representation: Feature Engineering

- Sparse Representation
  - Suppose that you had 1,000,000 different street names in your data set that you wanted to include as values for street_name.
  - Explicitly creating a binary vector of 1,000,000 elements where only 1 or 2 elements are true is a very inefficient representation in terms of both storage and computation time when processing these vectors.
  - In this situation, a common approach is to use a sparse representation in which only nonzero values are stored.
  - In sparse representations, an independent model weight is still learned for each feature value, as described before.

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*.