Chapter 3. Cross Validation / Regularization

Slides (pdf) Video (Panopto)

As we have discussed in the previous chapter, we must assess different model complexities. The metric used to determine the best model complexity cannot be the training error, as it will favor the model that overfits the training data, not the model that will perform the best in the future. However, we cannot use the test error either, because we cannot tamper with the test set until the very end. Otherwise, our test set no longer represents the "unknown".

3.1 The Validation Set

Since we cannot use the data that we trained on, nor the data that should be an unbiased representation of the future, we must designate a new set to determine the best model complexity that is neither trained on, nor promised to be an unbiased representation of future data. This will be the **validation set**.

Definition 3.0.1: Validation Set

The **validation set** is the set of the data used to optimize and find the best model for a task.

Validation data is not used to train on directly, but it is used at the end of each training round to calculate an error based on unseen data. This is different from the test set because the validation error is what we use to decide if training is worth continuing for a particular model, and how successful this model complexity is compared to the others.

Observe the process captured in pseudocode below:

```
train, validation, test = random_split(dataset)
for each model complexity p:
    model = train_model(model_p, train)
    val_err = error(model_p, validation)
    keep track of p with smallest val_err
return best p + error(model_best_p, test)
```

Figure 3.0.6: Pseudocode for validation set usage.

The total dataset is split into train, validation, and test sets. For each model complexity that we want to assess, we train a model of that complexity and calculate the validation error at the end of training. We only keep track of the lowest validation error because the model with the lowest validation error is the one that is most likely to be successful on future unseen data. This is how we define "best".

The pros of having a validation set are that it is easy to implement and it is relatively fast. Validation sets are used in Deep Learning for this reason. The cons of having a validation set are even though you don't train on validation data, you can still overfit to validation data since you keep tailoring your model to be successful on the validation set. Also, you have to sacrifice even more of your training data:

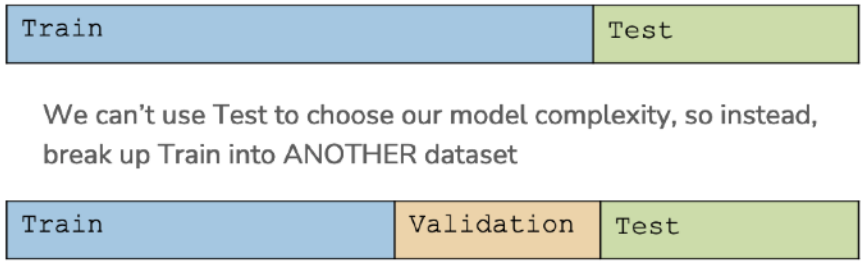

Figure 3.0.7: The validation set is taken from the training set, reducing the size of the training set.

3.2 Cross Validation

A clever idea to overcome the cons listed above is a technique called **cross validation**. Cross validation shuffles different parts of the training data for training versus validating.

Definition 3.0.2: Cross Validation

Cross Validation is a technique of resampling different portions of training data for validation on different iterations.

Here is the breakdown of the Cross Validation process:

1. First, the training set must be split into k number of chunks.

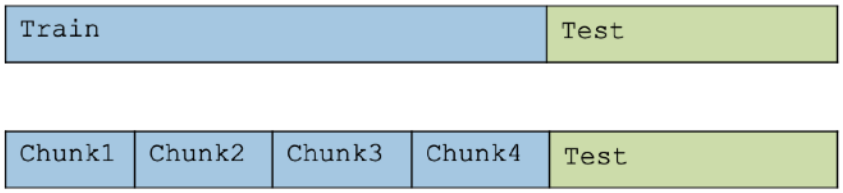

Figure 3.0.8: The training set is divided into a number of chunks.

2. One of these chunks is designated as a validation set for one iteration of training.
3. After one iteration of training, validation error is calculated on the single validation set chunk.
4. This is done k times, once for each chunk being the validation set.
5. All of the errors from each iteration are averaged. This is the final validation error for a model complexity.

Figure 3.0.9: The validation error is calculated by averaging the error from k different samples of validation data.

6. Repeat steps 1-5 for every model complexity you are assessing. The model complexity with the lowest average validation error is the best model.

Below we have provided pseudocode for the steps above:

```
train_set, test_set = random_split(dataset)
randomly shuffle train_set
choose a specific k and split it into k groups
for each model complexity p:                    *
    for i in [1, k]:
        model = train_model(model_p, chunks - i)
        val_err = error(model, chunk_i)
    avg_val_err = average val_err over chunks
    keep track of p with smallest avg_val_err
retrain the model with best complexity p on the
full training set
return the error of that model on the test set
```

Figure 3.0.10: Cross Validation pseudocode.

The pros of this method is that you don't get rid of any of your training data. Also, you prevent overfitting

because you train on a unique subset of the train set for each iteration, instead of the same training set for all iterations. The cons of this method is that it is very slow, because you have to train each model k many times. This makes training extremely computationally expensive. Your choice of k, how many chunks you split your train set into, also impacts your bias-variance trade-off: A lower k means you will have high bias while a higher k means you will have high variance. In practice, developers will use $k = 10$ because it achieves a fine medium balance.

Now, we will look at other ways to prevent overfitting.

There are two other main culprits behind overfitting: using too many features, and placing too much importance on specific features that are not that important. In other words, if the weight of a feature is very large, this could be a sign of overfitting. To overcome this, the model must "self-regulate" when its weights become too big. This process is called **regularization**.

3.3 Regularization

Definition 3.0.3: Regularization

Regularization is a technique used by a model to make sure it maintains balanced weights for its features to prevent overfitting.

Up until this point, we have always estimated out weights $\hat{w}$ by using a quality metric that minimizes the loss:

$$\hat{w} = \underset{w}{\operatorname{argmin}}\, L(w)$$

However, we have now learned that minimizing the loss for the training set might make our weights overfit the training set, and not be as ideal for future sets. So, we introduce a new term to our quality metric:

$$\hat{w} = \underset{w}{\operatorname{argmin}}\, L(w) + \lambda R(w)$$

where $R(w)$ is the magnitude of the weights and λ is our **regularization parameter.** We will thus account for our weights inside the quality metric. To account for the magnitude of all weights both negative and positive, we can either take the sum of all absolute values, or the sum of squares.

3.3.1 Ridge Regression

Definition 3.0.4: Ridge Regression

Ridge Regression is a regularization method that uses the sum of the squares of the weights in a model like so:

$$R(w) = |w_0|^2 + |w_1|^2 + ... + |w_d|^2 = ||w||_2^2 \tag{3.0.8}$$

The notation for Ridge Regression is $||w||_2^2$ because it uses an L2-norm, which is a type of norm. The notation for any norm uses the double bars and the subscript is the degree we raise each term to. Unlike a 2-norm we aren't raising the final sum to the $1/2$ power, so the norm is squared, hence it has the power of 2.

As Ridge Regression utilizes the sum of the square of our weights, our new quality metric becomes:

$$\hat{w} = \underset{w}{\text{argmin}}\, L(w) + \lambda ||w||_2^2$$

The regularization parameter, λ is a hyperparameter that determines how important the regularization term is in our quality metric.

if $\lambda = 0$:

$\Rightarrow$ $\hat{w} = \text{argmin}\ L(w)$

if $\lambda = \infty$:

$\Rightarrow$ Case 1: if $||w||_2^2 = 0$ then $\hat{w} = \text{argmin}\ L(w) + 0$

$\Rightarrow$ Case 2: if $||w||_2^2 > 0$ then $\hat{w} = \text{argmin}\ L(w) + \lambda||w||_2^2 \geq \lambda||w||_2^2 = \infty$. Since We are trying to minimize $L(w)$, $\text{min}L(w) = 0$ so $\hat{w} = 0$.

if λ is in between 0 and ∞:

$\Rightarrow$ $0 \leq ||w||_2^2 \leq \text{argmin}\ L(w)$

3.3.2 Coefficient Paths

Observe the change in size of the coefficients of the weights in our model as we increase λ. As λ increases, all weights get closer and closer to 0. This is what we mean by "regularizing" the weights.

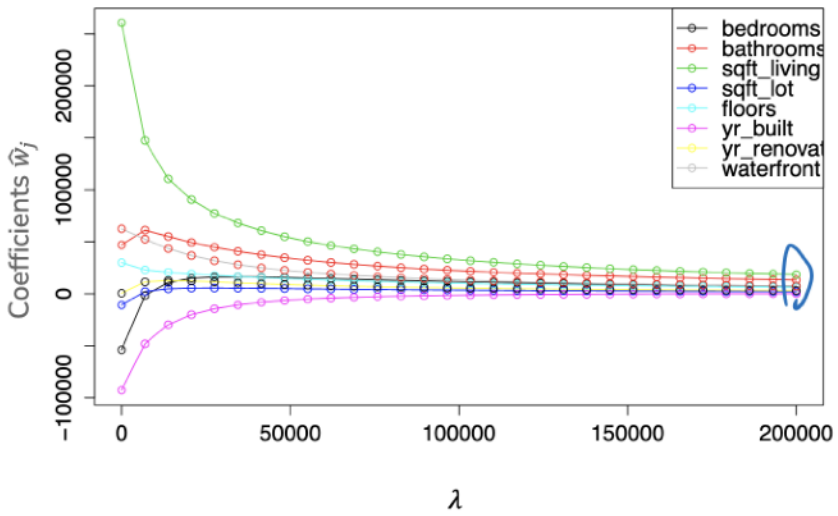

Figure 3.0.11: The effect of λ on the size of the weights.

3.3.3 The Intercept

When regularizing all the weights, we also happen to regularize w_0, the intercept term. This doesn't make sense because we shouldn't penalize a model for having a higher intercept since that may mean that the y value units are just really high. The intercept also doesn't affect the curvature of the a loss function. We deal with this by changing the measure of overfitting to not include the intercept:

$$\hat{w} = \underset{(w_0, w_{rest})}{\operatorname{argmin}} \; L(w_0, w_{rest}) + \lambda ||w_{rest}||_2^2$$

3.3.4 Scaling Features

Suppose we have a weight $\hat{w}_1$ for a feature, *house square footage*. What would happen if we changed the units of that feature to miles? Would $\hat{w}_1$ need to change? Since we change the input to be a of a larger scaled unit, the input values for this feature would be smaller. However, if the values are now smaller this means that we are giving the feature less importance in our model. The solution to this problem is **normalizing** all the features such that they are all on the same scale:

$$\tilde{h}_j(x^{(i)}) = \frac{h_j(x^{(i)}) - \mu_j(x^{(1)}, \ldots, x^{(N)})}{\sigma_j(x^{(1)}, \ldots, x^{(N)})}$$

Where

The mean of feature j:

$$\mu_j(x^{(1)}, \ldots, x^{(N)}) = \frac{1}{N} \sum_{i=1}^{N} h_j(x^{(i)})$$

The standard devation of feature j:

$$\sigma_j(x^{(1)}, \ldots, x^{(N)}) = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(h_j(x^{(i)}) - \mu_j(x^{(1)}, \ldots, x^{(N)})\right)^2}$$

Figure 3.0.12: We must apply this scaling to the test data and all future data using the means and standard deviations of the training set, otherwise the units of the model and the units of the data are not compatible.