Lesson 7

Weight Decay or L2 Regularization

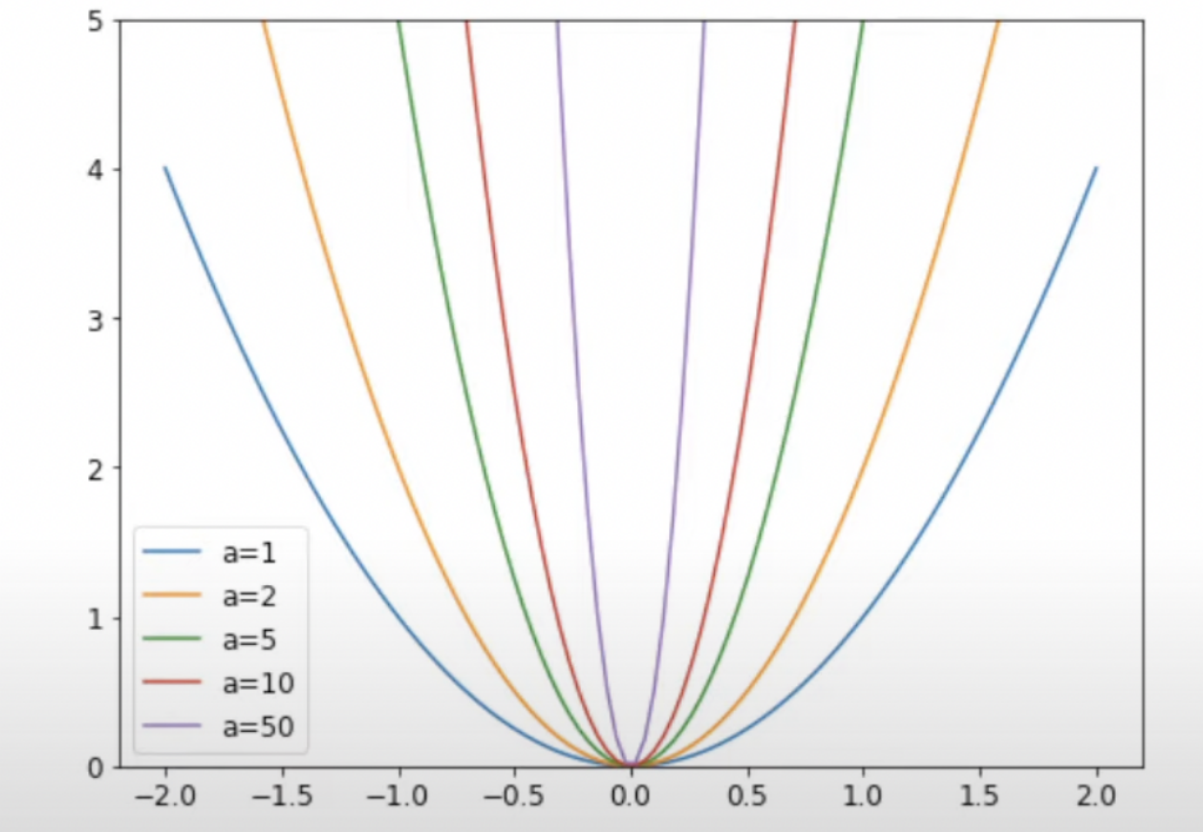
We alter the loss function by adding to it the sum of all the weights squared.

Why add weight to the loss?

Because adding weight to the loss will encourage the minimization of loss to minimize the weights as well.

Why do low weights prevent over fitting?

High weights allow functions to build sharp canyons for $y = ax^2$. Larger the a, narrower the parabola. Hence letting the model learn high parameters might allow the model to learn overly-complex functions that fit all the data points of the training data set and overfit. Reducing the ax^2 in the loss function will hinder the training of the model, but will results in models that generalize better.



Loss with weight decay:

```
loss_with_wd = loss + wd * (weight**2).sum()
```

In practice, calculating the loss is very inefficient, and the step can directly be utilized in the gradient:

```
weight.grad += wd * (2*weight)
# Since we define weight decay, we can make it twice as big
weight.grad += wd * weight
```

Chapter 9

- Continuous variables are numerical data such as age that is fed directly to the model because it can be added or multiplied directly.
- Categorical variables contain a number of discrete levels such as *movie id*, for which addition and multiplication don't have meaning

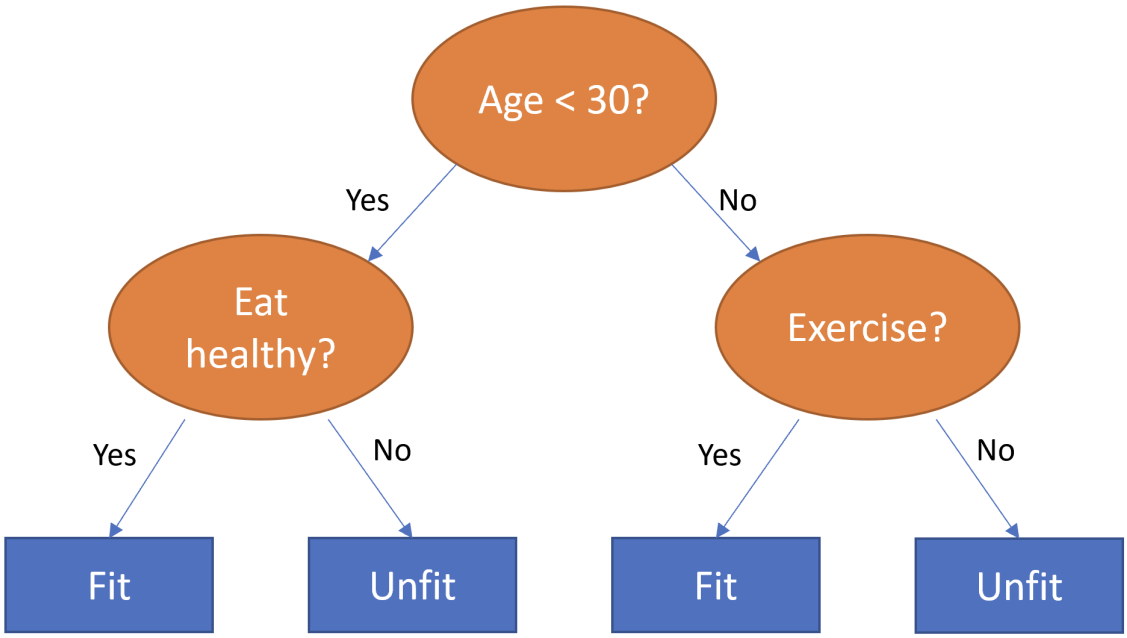
---

Usually its recommended to start with decision tree ensembles for tabular data, except when:

- High cardinal categorical variables (number of discrete levels representing categories, such as zip code)
- some columns contain plain text data, audio data, or image data

---

Decision trees ask a series of binary questions about the data.



1. Loop through each column of the dataset in turn.
2. For each column, loop through each possible level of that column in turn.
3. Try splitting the data into two groups, based on whether they are greater than or less than that value (or if it is a categorical variable, based on whether they are equal to or not equal to that level of that categorical variable).
4. Find the average sale price for each of those two groups, and see how close that is to the actual sale price of each of the items of equipment in that group. That is, treat this as a very simple "model" where our predictions are simply the average sale price of the item's group.
5. After looping through all of the columns and all the possible levels for each, pick the split point that gave the best predictions using that simple model.
6. We now have two different groups for our data, based on this selected split. Treat each of these as separate datasets, and find the best split for each by going back to step 1 for each group
7. Continue this process recursively, until you have reached some stopping criterion for each group—for instance, stop splitting a group further when it has only 20 items in it.

`add_datepart` creates a bunch of new date-related metadata columns such as year, month, week, day of week, day of month and so on.

`TabularProc` is a transform for pandas that transforms in place similar to transform for images. It also transforms once, instead of lazily. `TabularProc` is implemented using `TabularPandas` fastai class.

- `Categorify` replaces the column with a numerical categorical column
- `FillMissing` replaces missing values with the median of the column, and creates a new boolean column that is set to True for any row where the data was missing

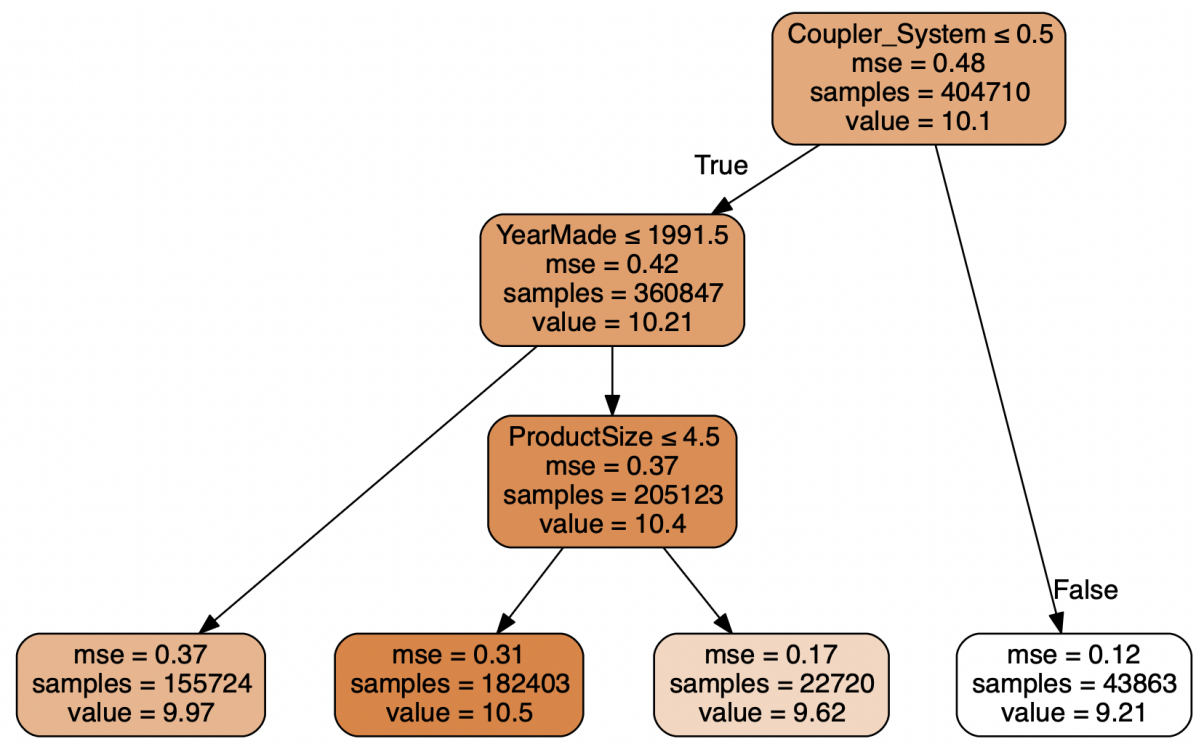
---

Random validation set is not appropriate for a time-based dataset. The test dataset is for two weeks after the train+validation. So the validation should be prepared for the same.

```
cond = (df.saleYear<2011) | (df.saleMonth<10)
train_idx = np.where( cond)[0]
valid_idx = np.where(~cond)[0]

splits = (list(train_idx),list(valid_idx))
```

Result



Random Forest and Decision Trees dont require encoding for categorical variables (ordinal categorical variables do!)

Bagging Predictors

1. Randomly choose a subset of the rows of your data (i.e., "bootstrap replicates of your learning set").
2. Train a model using this subset.
3. Save that model, and then return to step 1 a few times.
4. This will give you a number of trained models. To make a prediction, predict using all of the models, and then take the average of each of those model's predictions.

Because they are random subsets, the errors for the random subsets won't be correlated to each other. Because they are not correlated, the average of those errors will be zero! So the average of the models should give an accurate prediction of what we are trying to predict.

Random Forest: randomly select a subset of rows, and then for each split for each decision tree, selected from a random subset of columns

OOB Error (Out-of-Bag Error)

Using the rows in the training dataset that were not part of the training rows subset for that particular decision tree (so were unseen by that decision tree) as validation, and calculating the error on those rows.

OOB Error is then compared with the Validation error to find out if there are other causes of validation error other than expected — tells us how much we are *overfitting*.

```
r_mse(m.oob_prediction_, y)
```

Tree Variance for Prediction Confidence

We saw how the model averages the individual tree's predictions to get an overall prediction—that is, an estimate of the value. But how can we know the confidence of the estimate? One simple way is to use the standard deviation of predictions across the trees, instead of just the mean. This tells us the relative confidence of predictions. In general, we would want to be more cautious of

using the results for rows where trees give very different results (higher standard deviations), compared to cases where they are more consistent (lower standard deviations).

```
preds = np.stack([t.predict(valid_xs) for t in m.estimators_])
preds_std = preds.std(0)
preds_std[:5]
OUTPUT: array([0.25065395, 0.11043862, 0.08242067, 0.26988508, 0.15730173])
```

Feature Importance

The way these importances are calculated is quite simple yet elegant. The feature importance algorithm loops through each tree, and then recursively explores each branch. At each branch, it looks to see what feature was used for that split, and how much the model improves as a result of that split. The improvement (weighted by the number of rows in that group) is added to the importance score for that feature. This is summed across all branches of all trees, and finally the scores are normalized such that they add to 1.

```
def rf_feat_importance(m, df):
    return pd.DataFrame({'cols':df.columns, 'imp':m.feature_importances_}
                        ).sort_values('imp', ascending=False)
fi = rf_feat_importance(m, xs)
```

Partial Dependence Plots

Partial dependence plots try to answer the question: if a row varied on nothing other than the feature in question, how would it impact the dependent variable.

To answer this question, we can't just take the average sale price for each `YearMade`. The problem with that approach is that many other things vary from year to year as well, such as which products are sold, how many products have air-conditioning, inflation, and so forth. So, merely averaging over all the auctions that have the same `YearMade` would also capture the effect of how every other field also changed along with `YearMade` and how that overall change affected price.

Instead, what we do is replace every single value in the `YearMade` column with 1950, and then calculate the *predicted* sale price for every auction, and take the average over all auctions. Then we do the same for 1951, 1952, and so forth until our final year of 2011. This isolates the effect of only `YearMade` (even if it

does so by averaging over some imagined records where we assign a `YearMade` value that might never actually exist alongside some other values).

```
from sklearn.inspection import plot_partial_dependence

fig,ax = plt.subplots(figsize=(12, 4))
plot_partial_dependence(m, valid_xs_final, ['YearMade','ProductSize'],
                        grid_resolution=20, ax=ax);
```

Tree Interpreter

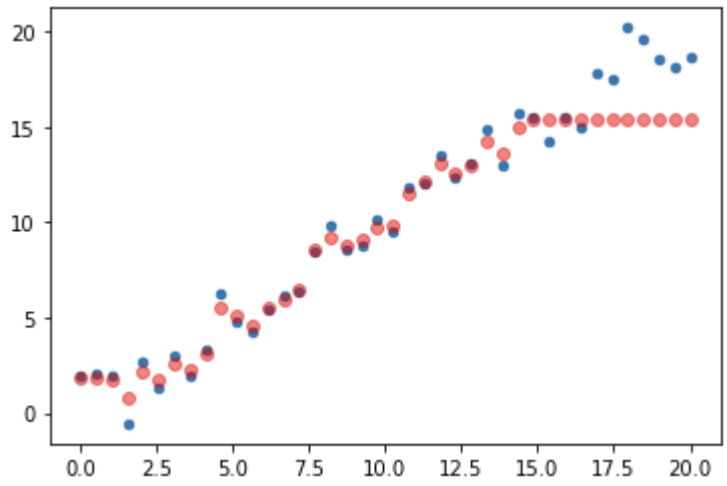
- Find out the reason for the confidence in a prediction

We have already seen how to compute feature importances across the entire random forest. The basic idea was to look at the contribution of each variable to improving the model, at each branch of every tree, and then add up all of these contributions per variable.

We can do exactly the same thing, but for just a single row of data. For instance, let's say we are looking at some particular item at auction. Our model might predict that this item will be very expensive, and we want to know why. So, we take that one row of data and put it through the first decision tree, looking to see what split is used at each point throughout the tree. For each split, we see what the increase or decrease in the addition is, compared to the parent node of the tree. We do this for every tree, and add up the total change in importance by split variable.

Extrapolation

Random forests are based on splitting of data on the bases of classes/avergaes of values that the tree has seen in the training data. Hence, random forests are not able to extrapolate outside of types of data it has already seen, and we need to make sure our validation set does not contain out of domain data.



We trained a random forest for the thirst 30 of 40 datapoints on x and their corresponding values on y. When asked to predict all of x, it chocked on the last 10 values, maxing out at the 30th value of y, because it had not seen the values of x before.

Finding out of domain data

We concatanate training and validation data, and create a new dependant variable 0 or 1 depending on if its training or validation. Then we train a new classifier; if we have any predictability on the new column then there is difference between traiing and validation datasets. We use feature importance to find out the columns that are significantly different between the two.

Neural Network Approach

We can create our TabularPandas object in the same way as when we created our random forest, with one very important addition: normalization. A random forest does not need any normalization—the tree building procedure cares only about the order of values in a variable, not at all about how they are scaled. But as we have seen, a neural network definitely does care about this. Therefore, we add the Normalize processor when we build our TabularPandas object.

Ensembling

A simple way to do ensembling is to average out the predictions between the RF and NN models

```
ens_preds = (to_np(preds.squeeze()) + rf_preds)/2
r_mse(ens_preds, valid_y)
```

This yielded a better result than either of the individual RF or NN models.

Boosting

- Train a small model that underfits your dataset.
- Calculate the predictions in the training set for this model.
- Subtract the predictions from the targets; these are called the "residuals" and represent the error for each point in the training set.
- Go back to step 1, but instead of using the original targets, use the residuals as the targets for the training.
- Continue doing this until you reach some stopping criterion, such as a maximum number of trees, or you observe your validation set error getting worse.

Using more trees in a random forest does not lead to overfitting, however, in a boosting ensemble, increasing the number of trees decreases the training error eventually causing over fitting.
