Aim: To predict price of bulldozer given the time series data of price.

Methodology:

Problem Definition:

Data: It's historical sales data of bulldozers. Including things like, model type, size, sale date and more.

There are 3 datasets:

Train.csv - Historical bulldozer sales examples up to 2011 (close to 400,000 examples with 50+ different attributes, including SalePrice which is the target variable).

Valid.csv - Historical bulldozer sales examples from January 1 2012 to April 30 2012 (close to 12,000 examples with the same attributes as Train.csv).

Test.csv - Historical bulldozer sales examples from May 1 2012 to November 2012 (close to 12,000 examples but missing the SalePrice attribute, as this is what we'll be trying to predict).

Evaluation

For this problem, as with many regression evaluations, the goal will be to get this value as low as possible.

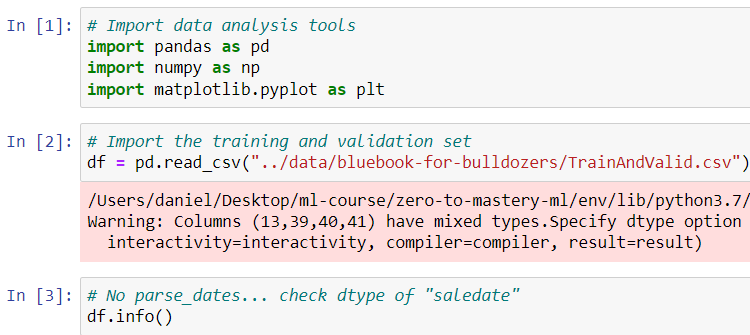
To see how well our model is doing, we'll calculate the RMSLE and then compare our results to others on the Kaggle leaderboard.

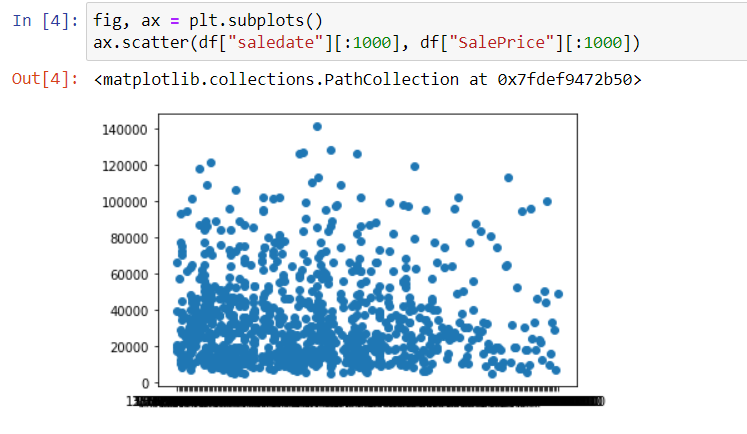
Features

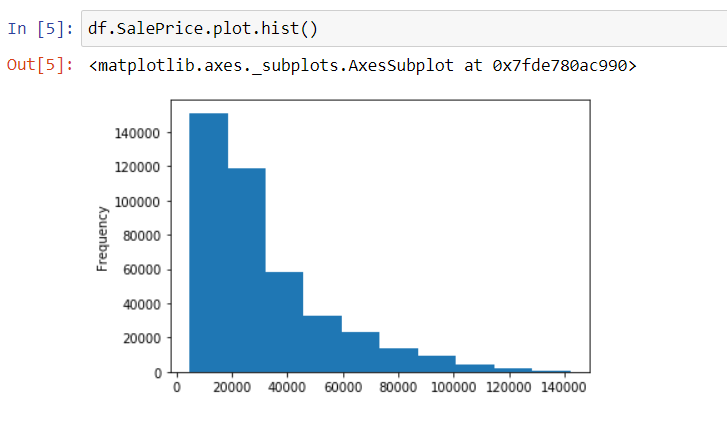
Features are different parts of the data.

First, we'll import the dataset and start exploring. Since we know the evaluation metric we're trying to minimize, our first goal will be building a baseline model and seeing how it stacks up against the competition.

Importing the data and preparing it for modelling

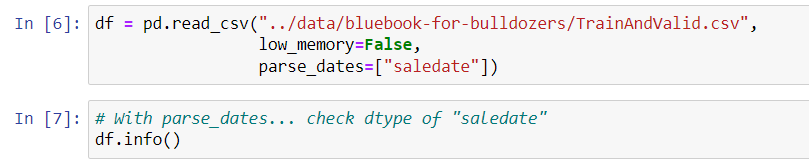


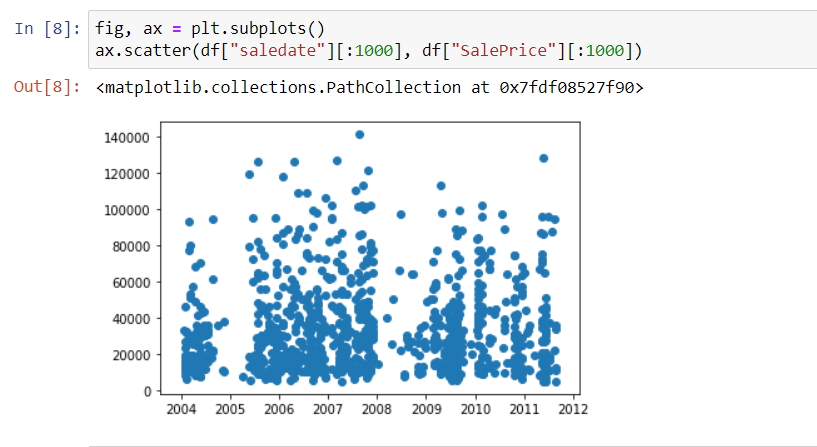




Parsing dates

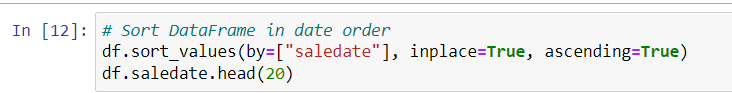
As we are working with time series data we will make sure date data is the format of a date time object.





Sort DataFrame by saledate

As we're working on a time series problem and trying to predict future examples given past examples, we will sort our data by date.

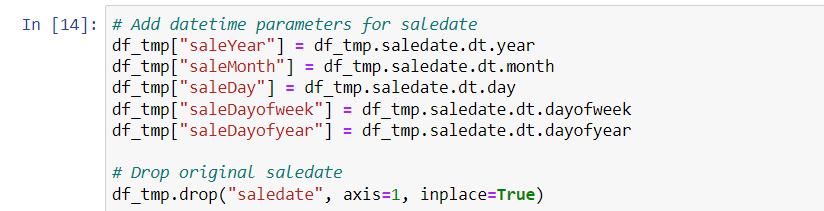


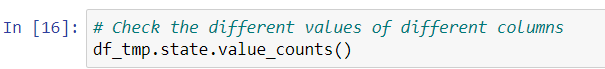
Make a copy of the original DataFrame

As we are manipulating data, we will make a new copy of it. This will keep the original Data Frame intact if we need it again.

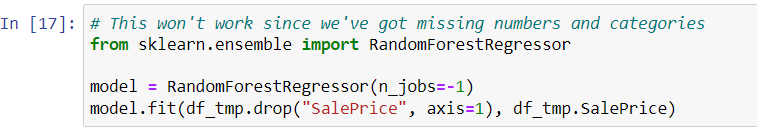


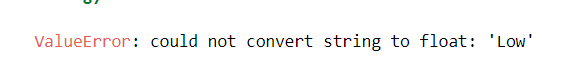
Add datetime parameters for saledate column

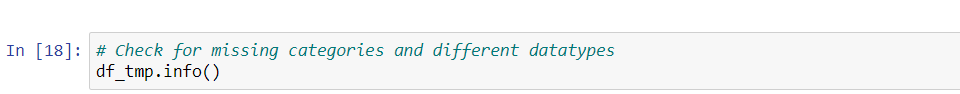


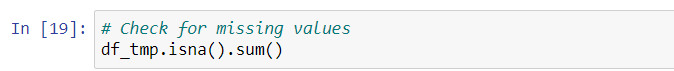


5. Modelling



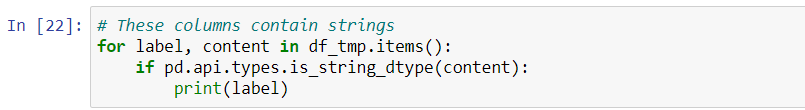


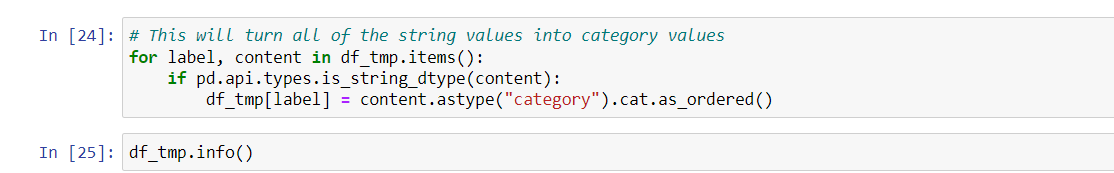




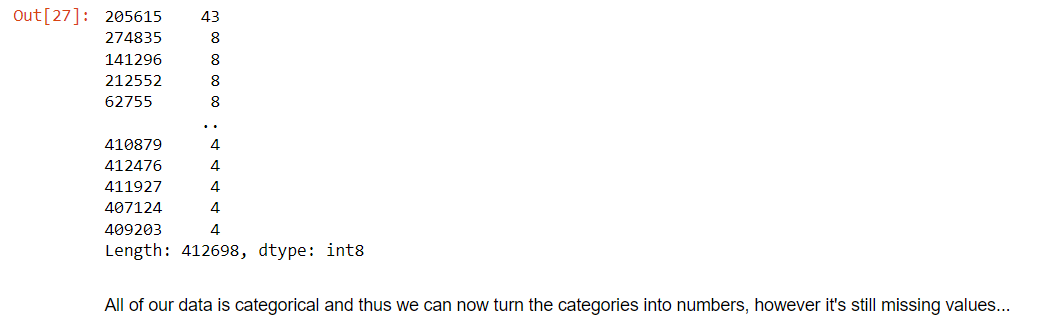
Convert strings to categories

We have converted all our columns with the string datatype into a category datatype.



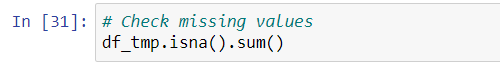
 











Fill missing values

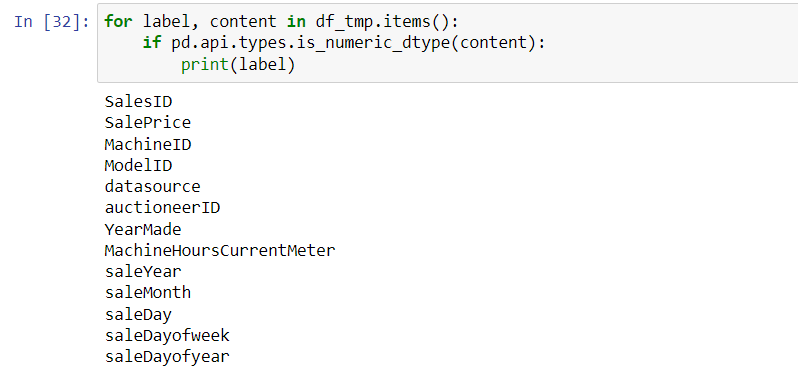
We know two things:

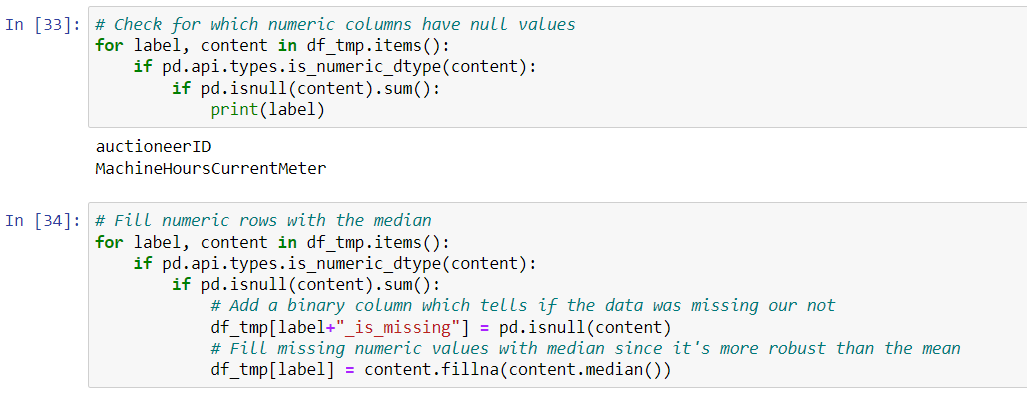
All of our data has to be numerical and there can't be any missing values.

And as we've seen using df\_tmp.isna().sum() our data still has plenty of missing values.

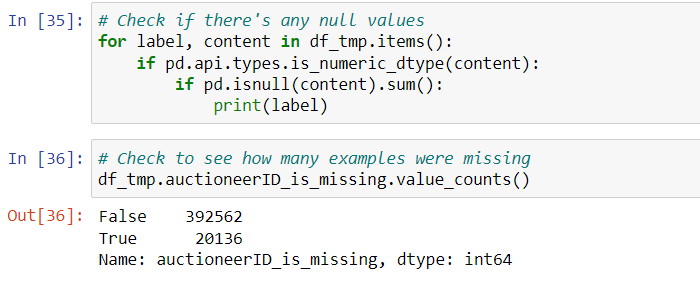
Filling Missing value

Filling numerical values first. We're going to fill any column with missing values with the median of that column.

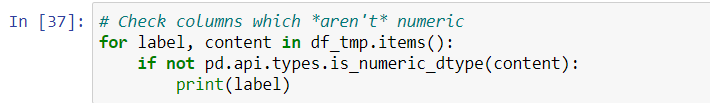


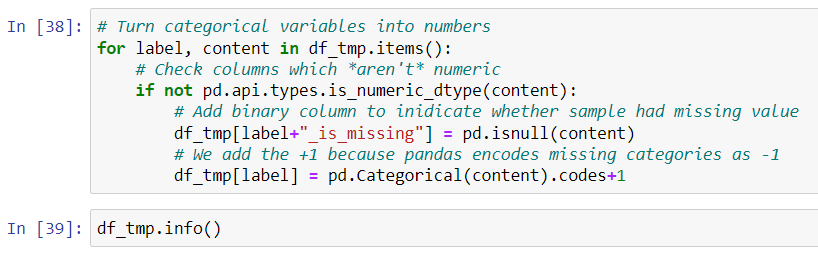


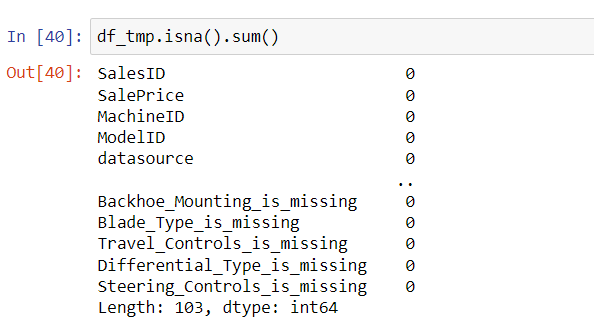
We can easily fill all of the missing numeric values in our dataset with the median. However, a numeric value may be missing for a reason. In other words, absence of evidence may be evidence of absence. Adding a binary column which indicates whether the value was missing or not helps to retain this information.



Filling and turning categorical variables to numbers







Splitting data into train/valid sets

Since we're working on a time series problem.

E.g. using past events to try and predict future events.

So, randomly splitting our data into train and test sets using something like train\_test\_split() wouldn't work.

Instead, we split our data into training, validation and test sets using the date each sample occured.

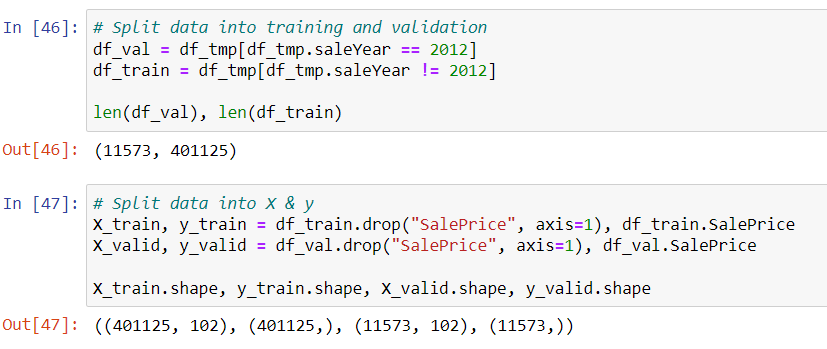
In our case:

Training = all samples up until 2011

Valid = all samples form January 1, 2012 - April 30, 2012

Test = all samples from May 1, 2012 - November 2012



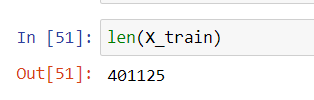


Building an evaluation function

Testing our model on a subset (to tune the hyperparameters)

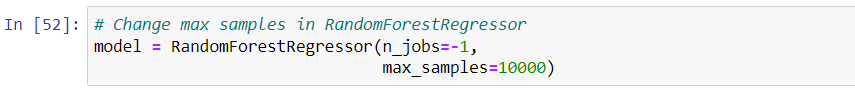
Retraing an entire model would take far too long to continuing experimenting as fast as we want to.

So what we'll do is take a sample of the training set and tune the hyperparameters on that before training a larger model.



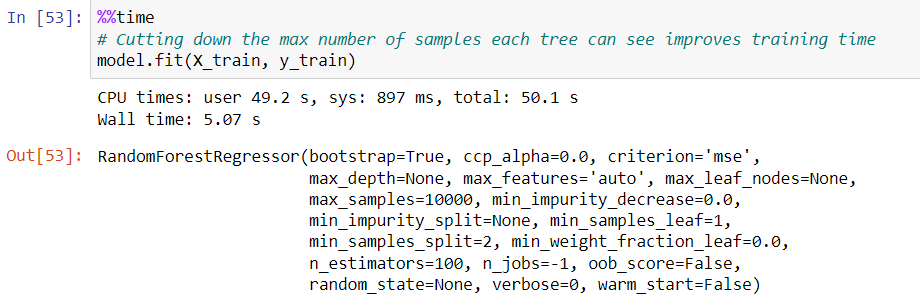
Depending on your computer, making calculations on ~400,000 rows may take a while.

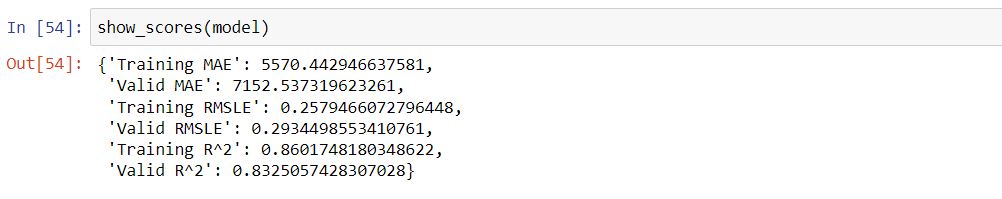
Let's alter the number of samples each n\_estimator in the RandomForestRegressor see's using the max\_samples parameter.



Setting max\_samples to 10000 means every n\_estimator (default 100) in our RandomForestRegressor will only see 10000 random samples from our DataFrame instead of the entire 400,000.

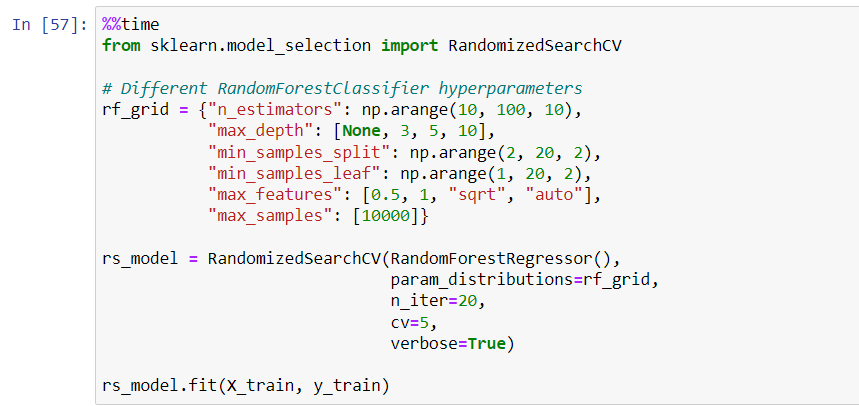
In other words, we'll be looking at 40x less samples which means we'll get faster computation speeds but we should expect our results to worsen (simple the model has less samples to learn patterns from).



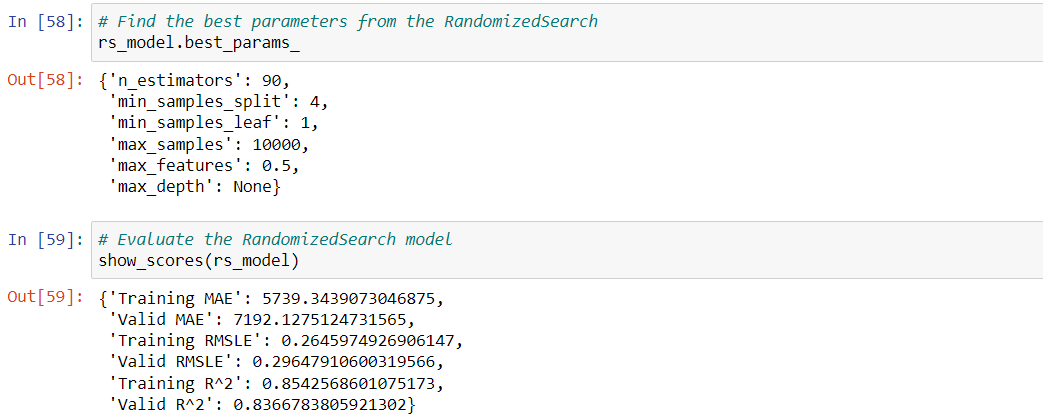


Hyperparameter tuning with RandomizedSearchCV

You can increase n\_iter to try more combinations of hyperparameters but in our case, we'll try 20 and see where it gets us.



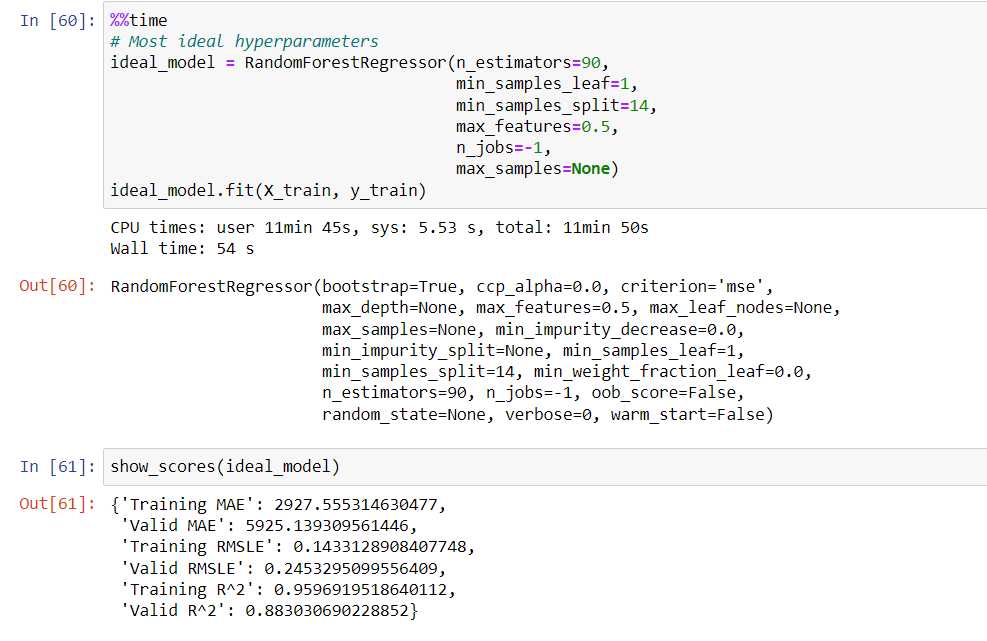




Train a model with the best parameters

In a model we prepared earlier, we tried 100 different combinations of hyperparameters (setting n\_iter to 100 in RandomizedSearchCV) and found the best results came from the ones you see below.

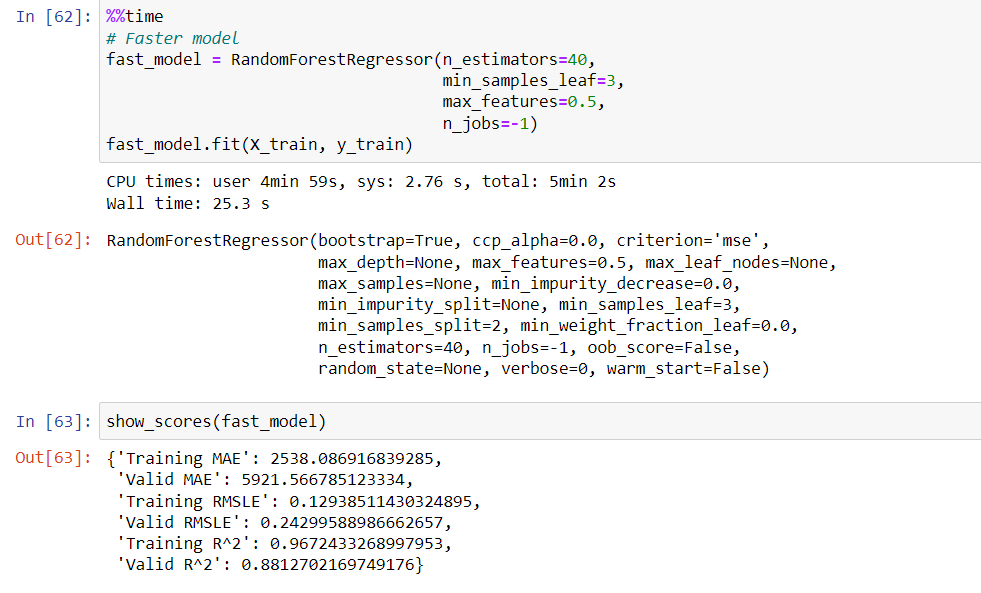
We'll instantiate a new model with these discovered hyperparameters and reset the max\_samples back to its original value.



With these new hyperparameters as well as using all the samples, we can see an improvement to our models performance.

We can make a faster model by altering some of the hyperparameters. Particularly by lowering n\_estimators since each increase in n\_estimators is basically building another small model.

However, lowering of n\_estimators or altering of other hyperparameters may lead to poorer results.

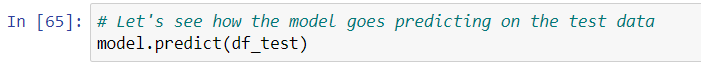


Make predictions on test data

Now we've got a trained model, it's time to make predictions on the test data.

Our model is trained on data prior to 2011. However, the test data is from May 1 2012 to November 2012.

So what we're doing is trying to use the patterns our model has learned in the training data to predict the sale price of a Bulldozer with characteristics it's never seen before but are assumed to be similar to that of those in the training data.





The test data isn't in the same format of our other data, so we have to fix it. Let's create a function to preprocess our data.

Preprocessing the data

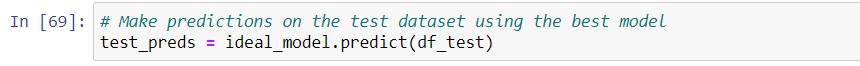
Our model has been trained on data formatted in the same way as the training data.

This means in order to make predictions on the test data, we need to take the same steps we used to preprocess the training data to preprocess the test data.

Remember: Whatever you do to the training data, you have to do to the test data.

Let's create a function for doing so (by copying the preprocessing steps we used above).

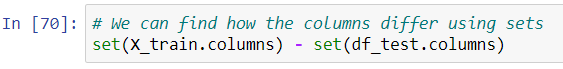






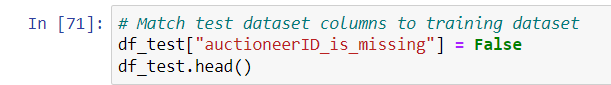
We've found an error and it's because our test dataset (after preprocessing) has 101 columns where as, our training dataset (X\_train) has 102 columns (after preprocessing).

Let's find the difference.



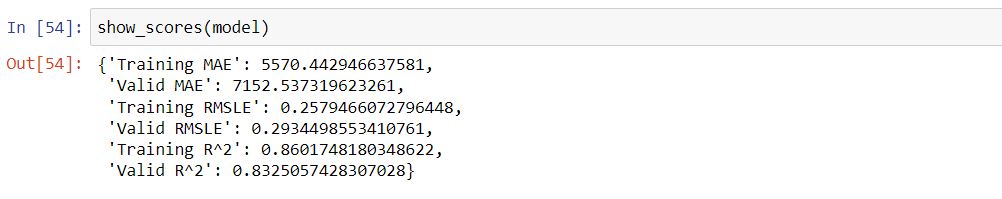
In this case, it's because the test dataset wasn't missing any auctioneerID fields.

To fix it, we'll add a column to the test dataset called auctioneerID\_is\_missing and fill it with False, since none of the auctioneerID fields are missing in the test dataset.



Conclusion:

Output without hyperparameter tuning:



Output after hyperparameter tuning:

