```
!pip install fastai
!pip install fastbook
!pip install dtreeviz
```

```
from fastbook import *
from pandas.api.types import is_string_dtype, is_numeric_dtype, is_categorical_dtype
from fastai.tabular.all import *
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from dtreeviz.trees import *
from IPython.display import Image, display_svg, SVG
from fastai import *

pd.options.display.max_rows = 20
pd.options.display.max_columns = 8
```

Tabular modeling

- takes data in the form of a table (like a spreadsheet or CSV). The objective is to predict the value in one column based on the values in the other columns.

What To Use ? ML or DL

First Approach : Dedision Trees Therefore, ensembles of decision trees are our first approach for analyzing a new tabular dataset. **Exception :** The exception to this guideline is when the dataset meets one of these conditions:

1. There are some high-cardinality categorical variables that are very important ("cardinality" refers to the number of discrete levels representing categories, so a high-cardinality categorical variable is something like a zip code, which can take on thousands of possible levels).
2. There are some columns that contain data that would be best understood with a neural network, such as plain text data.In practice, when we deal with datasets that meet these exceptional conditions, we always try both decision tree ensembles and deep learning to see which works best. It is likely that deep learning will be a useful approach in our example of collaborative filtering, as we have at least two high-cardinality categorical variables: the users and the movies. But in practice things tend to be less cut-and-dried, and there will often be a mixture of high- and low-cardinality categorical variables and continuous variables.

Entity Embedding

As entity embedding defines a distance measure for categorical variables it can be used for visualizing categorical data and for data clustering.

Categorial Embeddings

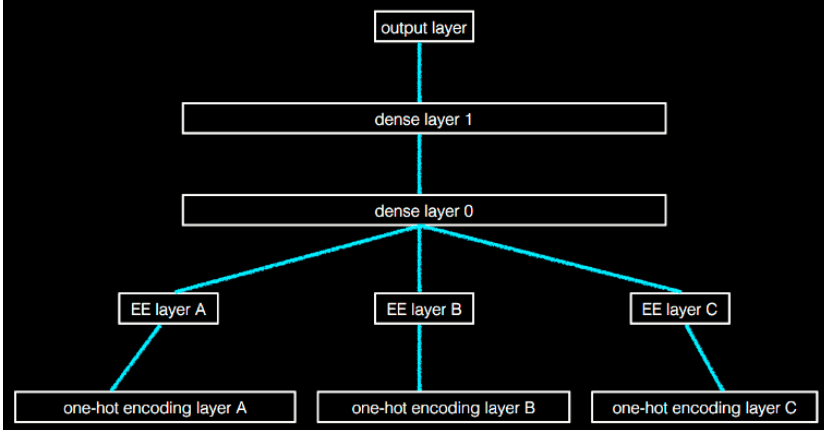
- continuous data - numercialdata can be directly fed into a model
- but **Categorial Data** (Qualitative Data) being discrete i.e movie id - can not be fed directly as addition,multiplication on them means nothing even if they are numeric
- one hot encoded input layer : method of convering categorial variables to fed them into the model

italicised text## Example Problem **Rossmann sales competition ran on Kaggle**

- **Paper Link** https://arxiv.org/abs/1604.06737
- **Their Analysis**

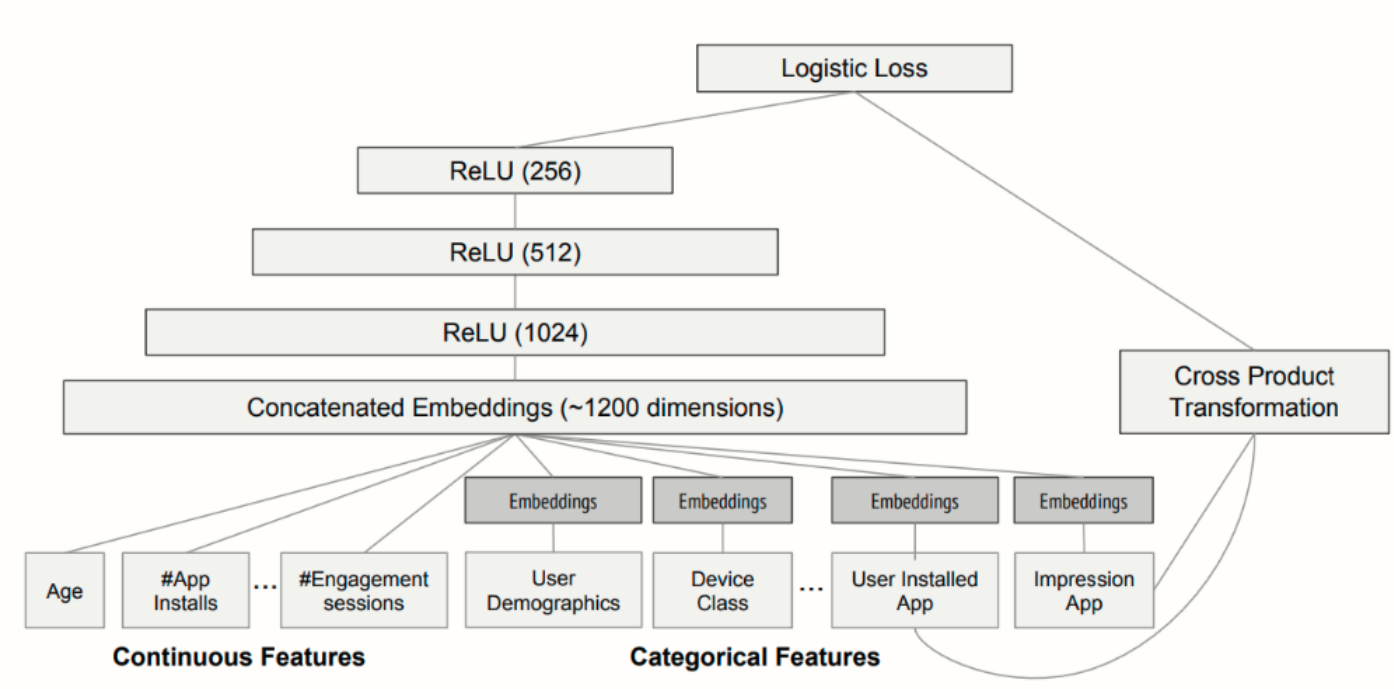
: Entity embedding not only reduces memory usage and speeds up neural networks compared with one-hot encoding, but more importantly by mapping similar values close to each other in the embedding space it reveals the intrinsic properties of the categorical variables... [It] is especially useful for datasets with lots of high cardinality features, where other methods tend to overfit... As entity embedding defines a distance measure for categorical variables it can be used for visualizing categorical data and for data clustering.

The paper also points out that (as we discussed in the last chapter) an embedding layer is exactly equivalent to placing an ordinary linear layer after every one-hot-encoded input layer. The authors used the diagram in <> to show this equivalence. Note that "dense layer" is a term with the same meaning as "linear layer," and the one-hot encoding layers represent inputs.



- **an embedding layer is exactly equivalent to placing an ordinary linear layer after every one-hot-encoded input layer.**

Another Example Paper

- **Google Play Store Recommendations**
  - https://arxiv.org/abs/1606.07792
  - 

What Was Used Before Deep Learning for This Categorial Data

Vast majority of datasets can be best modeled with just two methods:

- Ensembles of decision trees (i.e., random forests and gradient boosting machines), mainly for structured data (such as you might find in a database table at most companies)
- Multilayered neural networks learned with SGD (i.e., shallow and/or deep learning), mainly for unstructured data (such as audio, images, and natural language)

Double-click (or enter) to edit

Predicting Sales

sklearn -> for ML

- **Training Decision Trees**
- **Random Forests**

DataSet

- from kaggle competitions

```
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

```
Browse... No files selected.  Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving Test.csv to Test.csv
Saving TrainAndValid.csv to TrainAndValid.csv
User uploaded file "Test.csv" with length 3560907 bytes
User uploaded file "TrainAndValid.csv" with length 119791159 bytes
```

Look At The Data

Look at the Data

Kaggle provides information about some of the fields of our dataset. The Data explains that the key fields in *train.csv* are:

- `SalesID` :: The unique identifier of the sale.
- `MachineID` :: The unique identifier of a machine. A machine can be sold multiple times.
- `saleprice` :: What the machine sold for at auction (only provided in *train.csv*).
- `saledate` :: The date of the sale.

In any sort of data science work, it's important to *look at your data directly* to make sure you understand the format, how it's stored, what types of values it holds, etc. Even if you've read a description of the data, the actual data may not be what you expect. We'll start by reading the training set into a Pandas DataFrame. Generally it's a good idea to specify `low_memory=False` unless Pandas actually runs out of memory and returns an error. The `low_memory` parameter, which is `True` by default, tells Pandas to only look at a few rows of data at a time to figure out what type of data is in each column. This means that Pandas can actually end up using different data type for different rows, which generally leads to data processing errors or model training problems later.

Let's load our data and have a look at the columns:

```
df = pd.read_csv('TrainAndValid.csv', low_memory=False)
```

```
df.columns
```

```
Index(['SalesID', 'SalePrice', 'MachineID', 'ModelID', 'datasource',
       'auctioneerID', 'YearMade', 'MachineHoursCurrentMeter', 'UsageBand',
       'saledate', 'fiModelDesc', 'fiBaseModel', 'fiSecondaryDesc',
       'fiModelSeries', 'fiModelDescriptor', 'ProductSize',
       'fiProductClassDesc', 'state', 'ProductGroup', 'ProductGroupDesc',
       'Drive_System', 'Enclosure', 'Forks', 'Pad_Type', 'Ride_Control',
       'Stick', 'Transmission', 'Turbocharged', 'Blade_Extension',
       'Blade_Width', 'Enclosure_Type', 'Engine_Horsepower', 'Hydraulics',
       'Pushblock', 'Ripper', 'Scarifier', 'Tip_Control', 'Tire_Size',
       'Coupler', 'Coupler_System', 'Grouser_Tracks', 'Hydraulics_Flow',
       'Track_Type', 'Undercarriage_Pad_Width', 'Stick_Length', 'Thumb',
       'Pattern_Changer', 'Grouser_Type', 'Backhoe_Mounting', 'Blade_Type',
       'Travel_Controls', 'Differential_Type', 'Steering_Controls'],
      dtype='object')
```

```
df['ProductSize'].unique()
```

```
array([nan, 'Medium', 'Small', 'Large / Medium', 'Mini', 'Large', 'Compact'], dtype=object)
```

We can tell Pandas about a suitable ordering of these levels like so:

```
sizes = 'Large','Large / Medium','Medium','Small','Mini','Compact'
```

```
df['ProductSize'] = df['ProductSize'].astype('category') #astype -> converts it into categorical colum
df['ProductSize'].cat.set_categories(sizes, ordered=True, inplace=True)
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/arrays/categorical.py:2631: FutureWarning: The
  res = method(*args, **kwargs)
```

ordinal columns : At this point, a good next step is to handle ordinal columns. This refers to columns containing strings or similar, but where those strings have a natural ordering. For instance, here are the levels of ProductSize:

However, in this case Kaggle tells us what metric to use: root mean squared log error (RMSLE) between the actual and predicted auction prices. We need do only a small amount of processing to use this: we take the log of the prices, so that rmse of that value will give us what we ultimately need:

```
dep_var='SalePrice' # we are pridicting this (log of sale price)
```

```
df[dep_var]=np.log(df[dep_var])
```

```
df[dep_var]
```

```
0         11.097410
1         10.950807
2          9.210340
3         10.558414
4          9.305651
             ...
412693     9.210340
412694     9.259131
412695     9.433484
412696     9.210340
412697     9.472705
Name: SalePrice, Length: 412698, dtype: float64
```

Decision Tree

automatic procedure

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.

Handline Tabular Data Techniques

Using TabularPandas and TabularProc

A second piece of preparatory processing is to be sure we can handle strings and missing data. Out of the box, sklearn cannot do either. Instead we will use fastai's class `TabularPandas`, which wraps a Pandas DataFrame and provides a few conveniences. To populate a `TabularPandas`, we will use two `TabularProc`s, `Categorify` and `FillMissing`. A `TabularProc` is like a regular `Transform`, except that:

- It returns the exact same object that's passed to it, after modifying the object in place.
- It runs the transform once, when data is first passed in, rather than lazily as the data is accessed.

`Categorify` is a `TabularProc` that replaces a column with a numeric categorical column. `FillMissing` is a `TabularProc` that 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 value was missing. These two transforms are needed for nearly every tabular dataset you will use, so this is a good starting point for your data processing:

Tabular Pandas will also help us splitting into

- train , and
- validattion data sets for us

grabbing the data between dates - as per condition of question

Date

- Feature Engineering

```
df=add_datepart(df,'saledate')
```

adding in test set too

```
df_test = pd.read_csv('Test.csv', low_memory=False)
df_test = add_datepart(df_test, 'saledate')
```

```
' '.join(o for o in df.columns if o.startswith('sale'))
```

```
'saleYear saleMonth saleWeek saleDay saleDayofweek saleDayofyear saleIs_month_end saleIs_month_s
tart saleIs_quarter_end saleIs_quarter_start saleIs_year_end saleIs_year_start saleElapsed'
```

Using TabularPandas and TabularProc

```
#df = df.replace(np.nan, 0)
```

```
procs = [Categorify, FillMissing]
```

```
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))
```

```
train_idx.shape,valid_idx.shape
```

```
((404710,), (7988,))
```

Handling Categorical and Continuous Columns

- TabularPandas needs to be told which columns are continuous and which are categorical. We can handle that automatically using the helper function cont_cat_spli

```
# splits automatically into continuous , categorical variables
cont,cat = cont_cat_split(df, 1, dep_var=dep_var)
```

```
tabular = TabularPandas(df, procs, cat, cont, y_names=dep_var, splits=splits)
```

```
len(tabular.train),len(tabular.valid)
```

```
(404710, 7988)
```

```
tabular.show(3)
```

	UsageBand	fiModelDesc	fiBaseModel	fiSecondaryDesc	fiModelSeries	fiModelDescriptor	ProductSize	fiProductClassD
0	Low	521D	521	D	#na#	#na#	#na#	Wheel Loader - 1 to 120.0 Horsepo
1	Low	950FII	950	F	II	#na#	Medium	Wheel Loader - 1 to 175.0 Horsepo
2	High	226	226	#na#	#na#	#na#	#na#	Skid Steer Loac 1351.0 to 1601.(Operating Capa

However, the underlying items are all numeric:

```
tabular.items.head(3)
```

	SalesID	SalePrice	MachineID	ModelID	...	saleIs_year_start	saleElapsed	auctioneerID_na	MachineHoursCurrentMeter_na
0	1139246	11.097410	999089	3157	...	1	1.163635e+09	1	1
1	1139248	10.950807	117657	77	...	1	1.080259e+09	1	1
2	1139249	9.210340	434808	7009	...	1	1.077754e+09	1	1

3 rows × 67 columns

```
to1 = TabularPandas(df, procs, ['state', 'ProductGroup', 'Drive_System', 'Enclosure'],
                    [], y_names=dep_var, splits=splits)
to1.show(3)
```

	state	ProductGroup	Drive_System	Enclosure	SalePrice
0	Alabama	WL	#na#	EROPS w AC	11.097410
1	North Carolina	WL	#na#	EROPS w AC	10.950807
2	New York	SSL	#na#	OROPS	9.210340

```
to1.items[['state', 'ProductGroup', 'Drive_System', 'Enclosure']].head(3)
```

	state	ProductGroup	Drive_System	Enclosure
0	1	6	0	3
1	33	6	0	3
2	32	3	0	6

The conversion of categorical columns to numbers is done by simply replacing each unique level with a number. The numbers associated with the levels are chosen consecutively as they are seen in a column, so there's no particular meaning to the numbers in categorical columns after conversion. The exception is if you first convert a column to a Pandas ordered category (as we did for ProductSize earlier), in which case the ordering you chose is used. We can see the mapping by looking at the classes attribute:

```
tabular.classes['ProductSize']
```

```
['#na#', 'Large', 'Large / Medium', 'Medium', 'Small', 'Mini', 'Compact']
```

Creating the Decision Tree

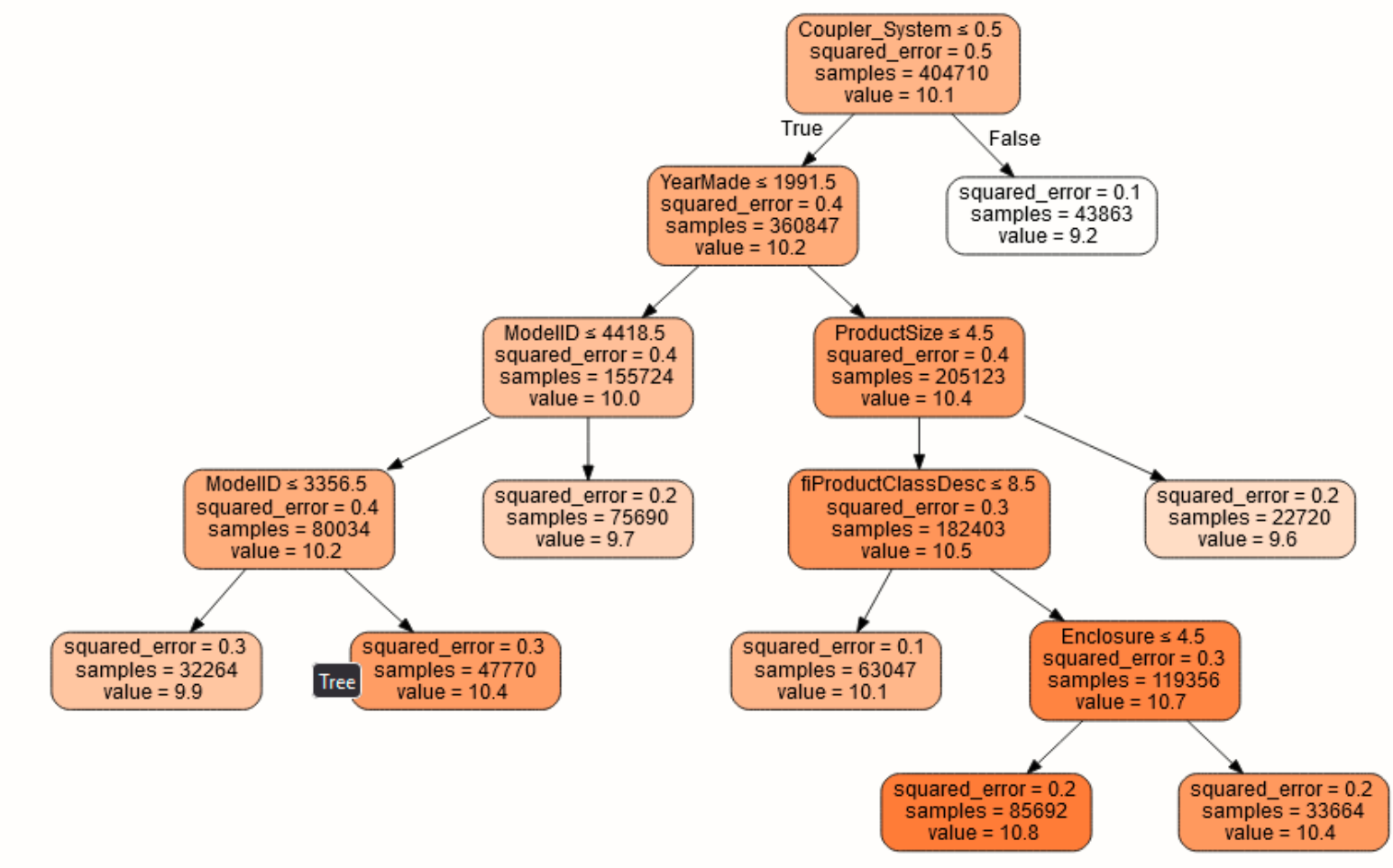
```
xs,y=tabular.train.xs,tabular.train.y
valid_xs,valid_y=tabular.valid.xs,tabular.valid.y
```

Now that our data is all numeric, and there are no missing values, we can create a decision tree:

To keep it simple, we've told sklearn to just create four leaf nodes.

```
m = DecisionTreeRegressor(max_leaf_nodes=8)
m.fit(xs, y);
```

```
draw_tree(m, xs, size=8, precision=1)
```



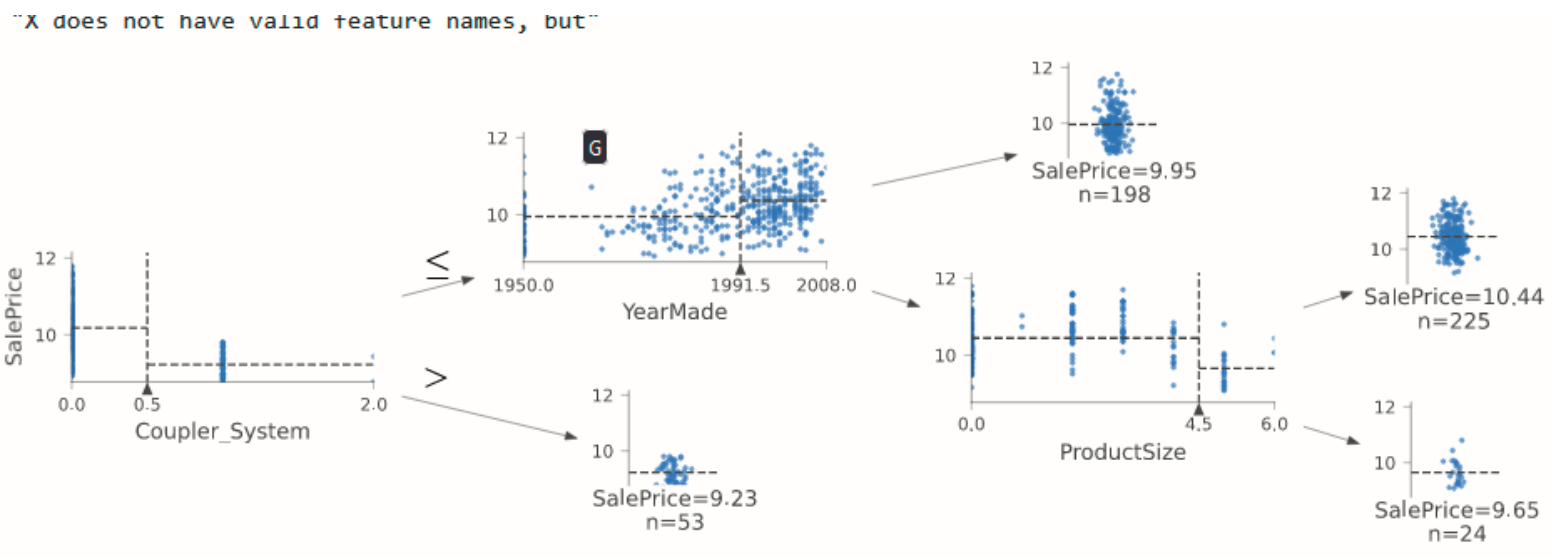
We can show the same information using Terence Parr's powerful dtreeviz library:

```
xs.loc[xs['YearMade']<1900, 'YearMade'] = 1950
valid_xs.loc[valid_xs['YearMade']<1900, 'YearMade'] = 1950
```

```
m = DecisionTreeRegressor(max_leaf_nodes=4).fit(xs, y)

dtreeviz(m, xs.iloc[samp_idx], y.iloc[samp_idx], xs.columns, dep_var,
         fontname='DejaVu Sans', scale=1.6, label_fontsize=10,
```

```
        orientation='LR')
```



function to check the root mean squared error of our model (m_rmse),

```
def r_mse(pred,y): return round(math.sqrt(((pred-y)**2).mean()), 6)
def m_rmse(m, xs, y): return r_mse(m.predict(xs), y)
```

Categorial Variables

- In a decision tree, we don't have embeddings layers—so how can these untreated categorical variables do anything useful in a decision tree? For instance, how could something like a product code be used?
- The short answer is: it just works! Think about a situation where there is one product code that is far more expensive at auction than any other one. In that case, any binary split will result in that one product code being in some group, and that group will be more expensive than the other group. Therefore, our simple decision tree building algorithm will choose that split. Later during training the algorithm will be able to further split the subgroup that contains the expensive product code, and over time, the tree will home in on that one expensive product.
- It is also possible to use one-hot encoding to replace a single categorical variable with multiple one-hot-encoded columns, where each column represents a possible level of the variable. Pandas has a get_dummies method which does just that.

However, there is not really any evidence that such an approach improves the end result. So, we generally avoid it where possible, because it does end up making your dataset harder to work with.

Random Forests

In 2001 Leo Breiman went on to demonstrate that this approach to building models, when applied to

decision tree building algorithms, was particularly powerful. He went even further than just randomly choosing rows for each model's training, but also randomly selected from a subset of columns when choosing each split in each decision tree. He called this method the random forest.

one liner :In essence a random forest is a model that averages the predictions of a large number of decision trees, which are generated by randomly varying various parameters that specify what data is used to train the tree and other tree parameters. Bagging is a particular approach to "ensembling," or combining the results of multiple models together.

Property :

- not sensitive to the hyperparameter choices

> One of the most important properties of random forests is that they aren't very sensitive to the hyperparameter choices, such as `max_features`. You can set `n_estimators` to as high a number as you have time to train—the more trees you have, the more accurate the model will be. `max_samples` can often be left at its default, unless you have over 200,000 data points, in which case setting it to 200,000 will make it train faster with little impact on accuracy. `max_features=0.5` and `min_samples_leaf=4` both tend to work well, although sklearn's defaults work well too.

Begging Predictors

- paper link https://www.stat.berkeley.edu/~breiman/bagging.pdf
- Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. The aggregation averages over the versions... The multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets. Tests… show that bagging can give substantial gains in accuracy

If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.

> Here is the procedure that Breiman is proposing:
>
> 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.

**More You Train Better it will be **

- This procedure is known as "bagging." It is based on a deep and important insight: although each of the models trained on a subset of data will make more errors than a model trained on the full dataset, those errors will not be correlated with each other. Different models will make different errors. The average of those errors, therefore, is: zero! So if we take the average of all of the models' predictions, then we should end up with a prediction that gets closer and closer to the correct answer, the more models we have. This is an extraordinary result—it means that we can improve the accuracy of nearly any kind of machine learning algorithm by training it multiple times, each time on a different random subset of the data, and averaging its predictions.

```
# n_estimator -> number of decision trees we want
# max_samples -> number of rows for training each tree
# max_features =0.5-> takes half of the total number of cols
# stoping creteria -> min_samples_leaf
# n_jobs-> use all CPU to build trees in parallel
def rf(xs, y, n_estimators=40, max_samples=200_000,
       max_features=0.5, min_samples_leaf=5, **kwargs):
    return RandomForestRegressor(n_jobs=-1, n_estimators=n_estimators,
        max_samples=max_samples, max_features=max_features,
        min_samples_leaf=min_samples_leaf, oob_score=True).fit(xs, y)
# create a random forest and fit it to x,y
```

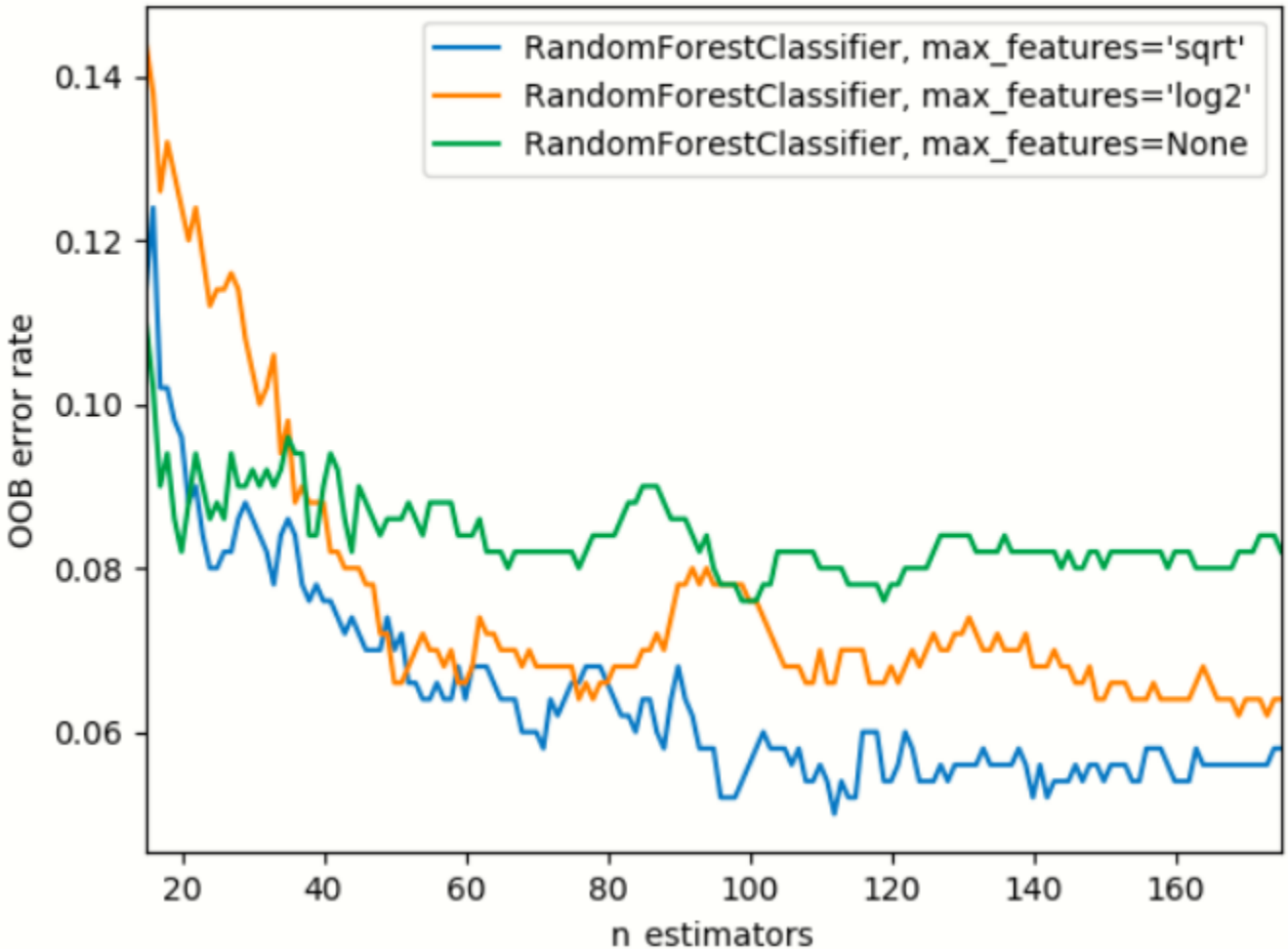
```
m=rf(xs,y)
```

Our validation RMSE is now much improved over our last result produced by the DecisionTreeRegressor, which made just one tree using all the available data:

```
m_rmse(m, xs, y), m_rmse(m, valid_xs, valid_y)
```

```
(0.170968, 0.232982)
```

Effects of Different Estimators - max_features



Seeing The Impact on our Forest

```
# array having predictions in each individual tree (rows in data)
preds=np.stack([t.predict(valid_xs) for t in m.estimators_])
```

Result Prediction for a radnom subset (begging)

```
m_rmse(m,xs,y),r_mse(preds.mean(0),valid_y)
```

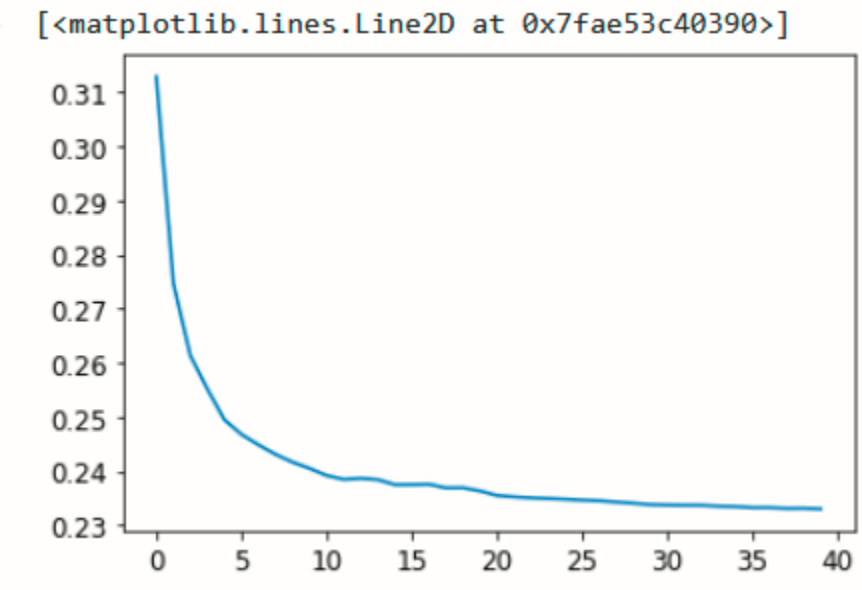
```
(0.170968, 0.232982)
```

- valid set is worse than training set

Behavior of RMSE as more trees added

```
plt.plot([      r_mse(preds[:i+1].mean(0),
                valid_y)for i in range(40)    ])
# 40 estimators - prediction of inidvidual trees
```

```
[<matplotlib.lines.Line2D at 0x7fae53c40390>]
```



Overfitting Occured

The performance on our validation set is worse than on our training set. But is that because we're overfitting, or because the validation set covers a different time period, or a bit of both? With the existing information we've seen, we can't tell. However, random forests have a very clever trick called out-of-bag (OOB) error that

can help us with this (and more!).

Trick : **Out of Bag Error**

> Recall that in a random forest, each tree is trained on a different subset of the training data. The OOB error is a way of measuring prediction error on the training set by only including in the calculation of a row's error trees where that row was *not* included in training. This allows us to see whether the model is overfitting, without needing a separate validation set.
>
> > A: My intuition for this is that, since every tree was trained with a different randomly selected subset of rows, out-of-bag error is a little like imagining that every tree therefore also has its own validation set. That validation set is simply the rows that were not selected for that tree's training.
>
> This is particularly beneficial in cases where we have only a small amount of training data, as it allows us to see whether our model generalizes without removing items to create a validation set. The OOB predictions are available in the `oob_prediction_` attribute. Note that we compare them to the training labels, since this is being calculated on trees using the training set.

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

```
(0.170968, 0.210685)
```

- **we get sense of how much we are overfitting without needing a separate validation set**
- it considers the whole training set not the last weeks

much less than having a common validation set

Model Interpretation

For tabular data, model interpretation is particularly important. For a given model, the things we are most likely to be interested in are:

1. How confident are we in our predictions using a particular row of data?
2. For predicting with a particular row of data, what were the most important factors, and how did they influence that prediction?
3. Which columns are the strongest predictors, which can we ignore?
4. Which columns are effectively redundant with each other, for purposes of prediction?
5. How do predictions vary, as we vary these columns?

Parameters

1. **confidence on predictions**
2. **impacting factors and their affect**
3. **particular cols importance**
4. **detecting redundance/ similarity in cols**
5. **effect of varying cols on precitions**

1. Tree Variance for Prediction Confidence

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.shape
```

```
(40, 7988)
```

```
preds_std=preds.std(0)
```

```
preds_std  # how much the trees vary
```

```
array([0.21851452, 0.08835592, 0.09955633, ..., 0.16396675, 0.1196267 , 0.1196267 ])
```

As you can see, the confidence in the predictions varies widely. For some auctions, there is a low standard deviation because the trees agree. For others it's higher, as the trees don't agree. This is information that would be useful in a production setting; for instance, if you were using this model to decide what items to bid on at auction, a low-confidence prediction might cause you to look more carefully at an item before you made a bid.

Feature Importance

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)
fi
```

	cols	imp

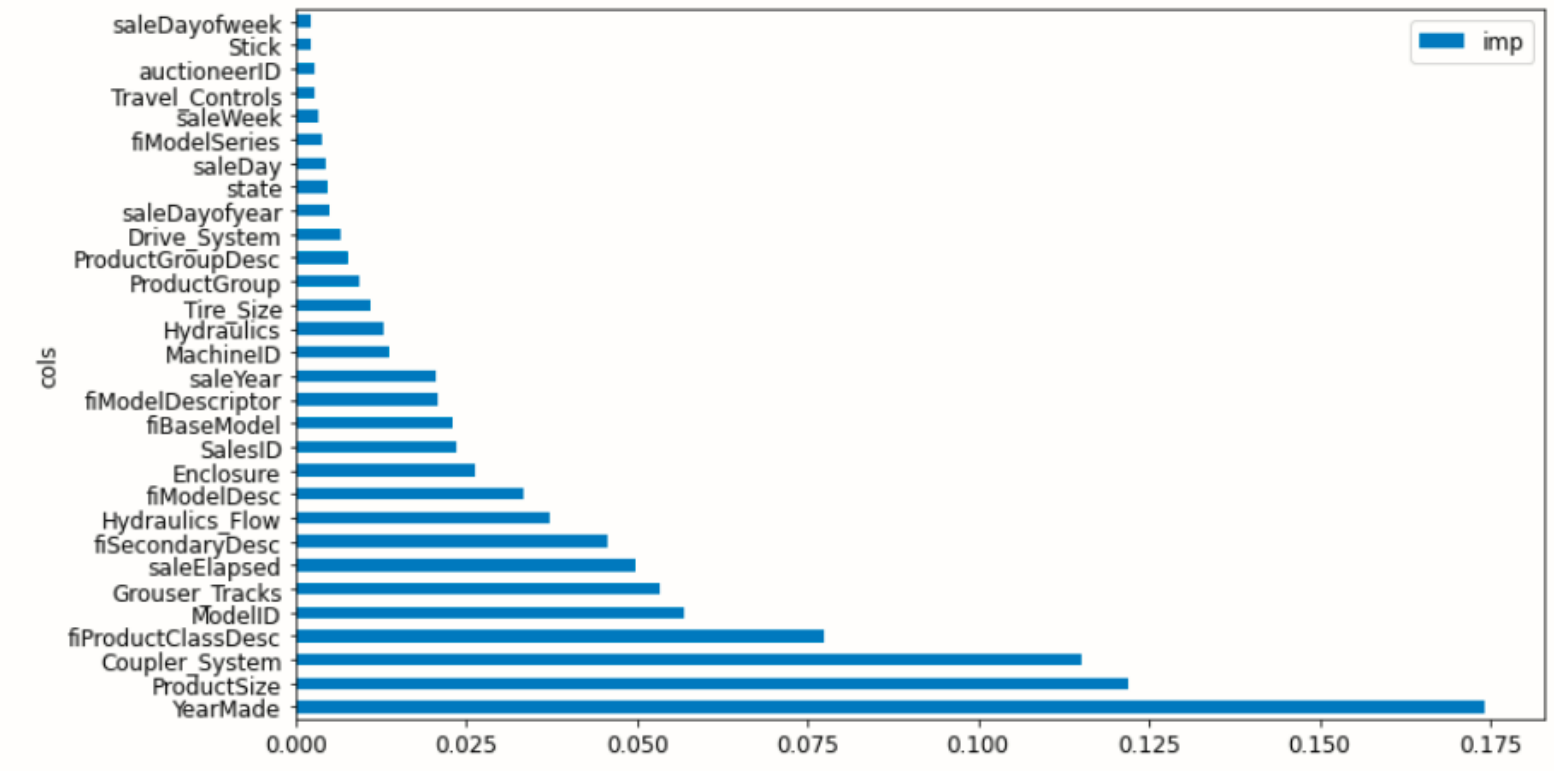
57	YearMade	0.174262
6	ProductSize	0.122020
30	Coupler_System	0.115190
7	fiProductClassDesc	0.077410
54	ModelID	0.056907
...	...	...
45	saleIs_month_start	0.000015
46	saleIs_quarter_end	0.000013
47	saleIs_quarter_start	0.000004
48	saleIs_year_end	0.000000
49	saleIs_year_start	0.000000

66 rows × 2 columns

A plot of the feature importances shows the relative importances more clearly:

```
def plot_fi(fi):
    return fi.plot('cols', 'imp', 'barh', figsize=(12,7), legend=true)

plot_fi(fi[:30]);
```



3. Removing low-importance variables

```
to_keep=fi[fi.imp>0.005].cols
len(to_keep)
```

```
21
```

only left wiht 21 cols

now Retrain the model

```
xs_imp = xs[to_keep]
valid_xs_imp = valid_xs[to_keep]
```

```
m = rf(xs_imp, y)
```

```
m_rmse(m, xs_imp, y), m_rmse(m, valid_xs_imp, valid_y)
```

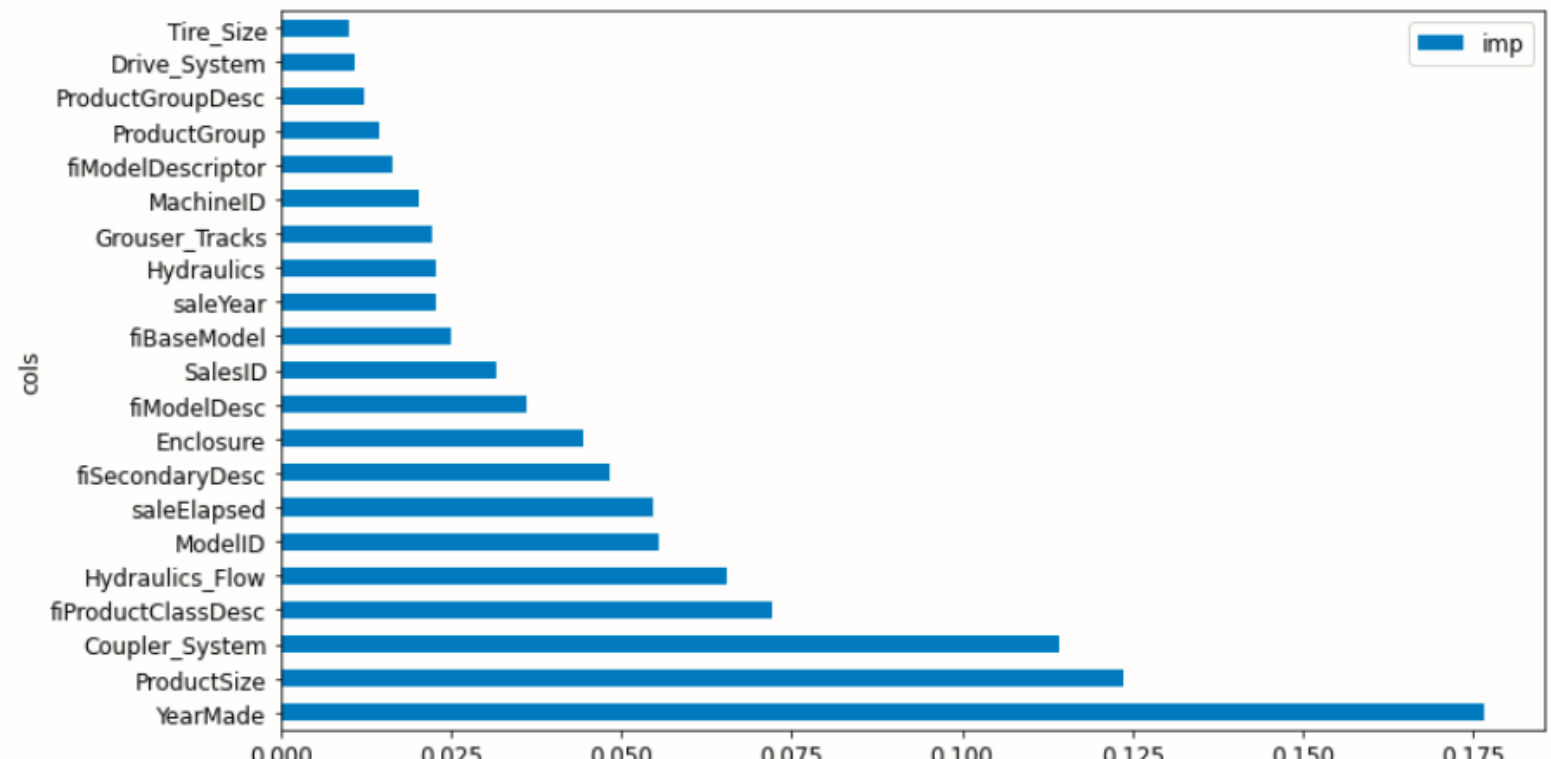
```
(0.181269, 0.230329)
```

Our accuracy is about the same, but we have far fewer columns to study:

```
len(xs.columns), len(xs_imp.columns)
```

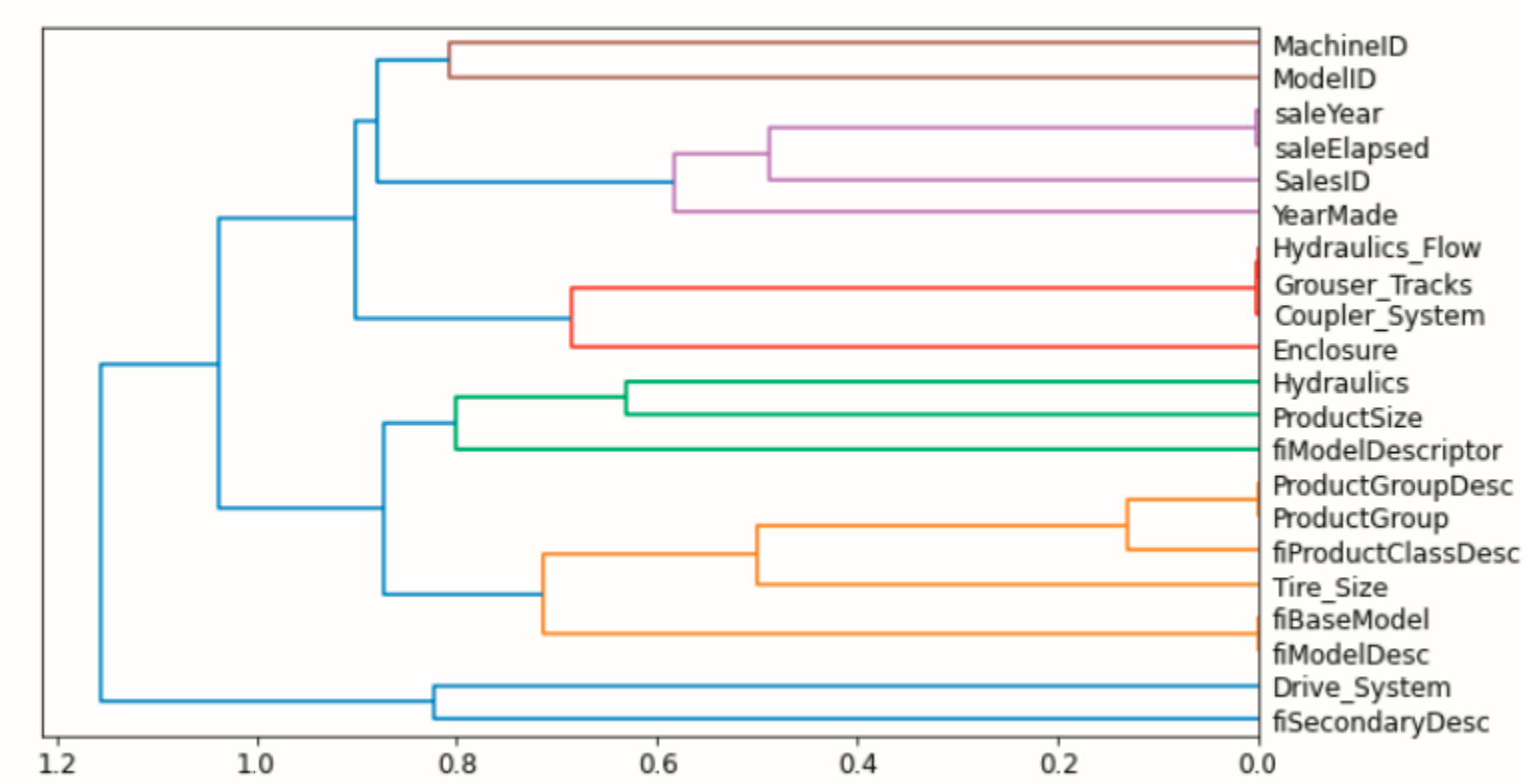
```
(66, 21)
```

```
plot_fi(rf_feat_importance(m, xs_imp));
```



4. Removing Redundancy

```
cluster_columns(xs_imp)
```



In this chart, the pairs of columns that are most similar are the ones that were merged together early, far from the "root" of the tree at the left. Unsurprisingly, the fields ProductGroup and ProductGroupDesc were merged quite early, as were saleYear and saleElapsed and fiModelDesc and fiBaseModel. These might be so closely correlated they are practically synonyms for each other.

removing some of these closely related features to see if the model can be simplified without impacting the accuracy

```
def get_oob(df):
    m = RandomForestRegressor(n_estimators=40, min_samples_leaf=15,
        max_samples=50000, max_features=0.5, n_jobs=-1, oob_score=True)
    m.fit(df, y)
    return m.oob_score_
```

```
get_oob(xs_imp)
```

```
0.8774322737593935
```

Manually Removing Redundant Cols

```
{c:get_oob(xs_imp.drop(c, axis=1)) for c in (
    'saleYear', 'saleElapsed', 'ProductGroupDesc','ProductGroup',
    'fiModelDesc', 'fiBaseModel',
    'Hydraulics_Flow','Grouser_Tracks', 'Coupler_System')}
```

```
 {'Coupler_System': 0.8777584797729616,
  'Grouser_Tracks': 0.8776388823356103,
  'Hydraulics_Flow': 0.8772171225797685,
  'ProductGroup': 0.8774327022343623,
  'ProductGroupDesc': 0.8781045430391307,
  'fiBaseModel': 0.8765382013702481,
  'fiModelDesc': 0.8759816280640887,
  'saleElapsed': 0.8724883617288165,
  'saleYear': 0.8766278061967991}
```

We'll drop one from each of the tightly aligned pairs we noticed earlier

```
to_drop = ['saleYear', 'ProductGroupDesc', 'fiBaseModel', 'Grouser_Tracks']
get_oob(xs_imp.drop(to_drop, axis=1))
```

```
 0.8744712309409295
```

Accuracy still remains almost same Looking good! This is really not much worse than the model with all the fields.

Let's create DataFrames without these columns

```
xs_final = xs_imp.drop(to_drop, axis=1)
valid_xs_final = valid_xs_imp.drop(to_drop, axis=1)
```

```
m = rf(xs_final, y)
m_rmse(m, xs_final, y), m_rmse(m, valid_xs_final, valid_y)
```

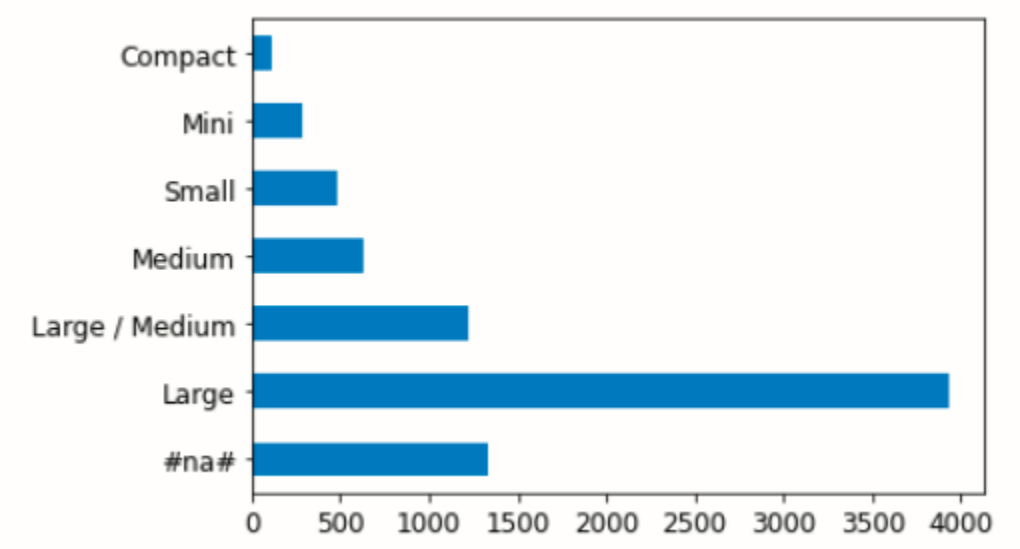
```
 (0.183516, 0.233244)
```

By focusing on the most important variables, and removing some redundant ones, we've greatly simplified our model. Now, let's see how those variables affect our predictions using partial dependence plots.

5. Partial Dependence

As we've seen, the two most important predictors are ProductSize and YearMade. We'd like to understand the relationship between these predictors and sale price.

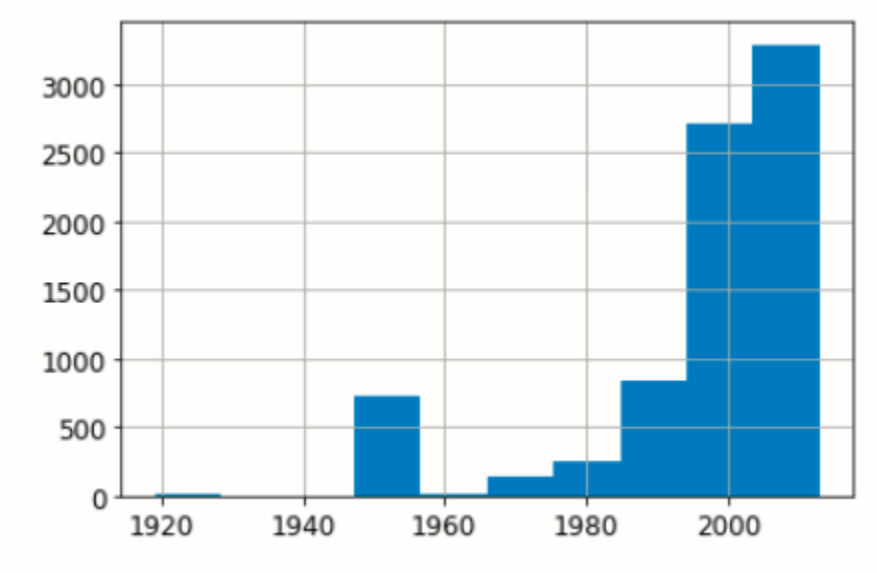
```
p = valid_xs_final['ProductSize'].value_counts(sort=False).plot.barh()
c = tabular.classes['ProductSize']
plt.yticks(range(len(c)), c);
```



The largrest group is #na#, which is the label fastai applies to missing values.

Let's do the same thing for YearMade. Since this is a numeric feature, we'll need to draw a histogram, which groups the year values into a few discrete bins:

```
ax = valid_xs_final['YearMade'].hist()
```



Other than the special value 1950 which we used for coding missing year values, most of the data is from after 1990.

Now we're ready to look at *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?

For instance, how does `YearMade` impact sale price, all other things being equal?

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.

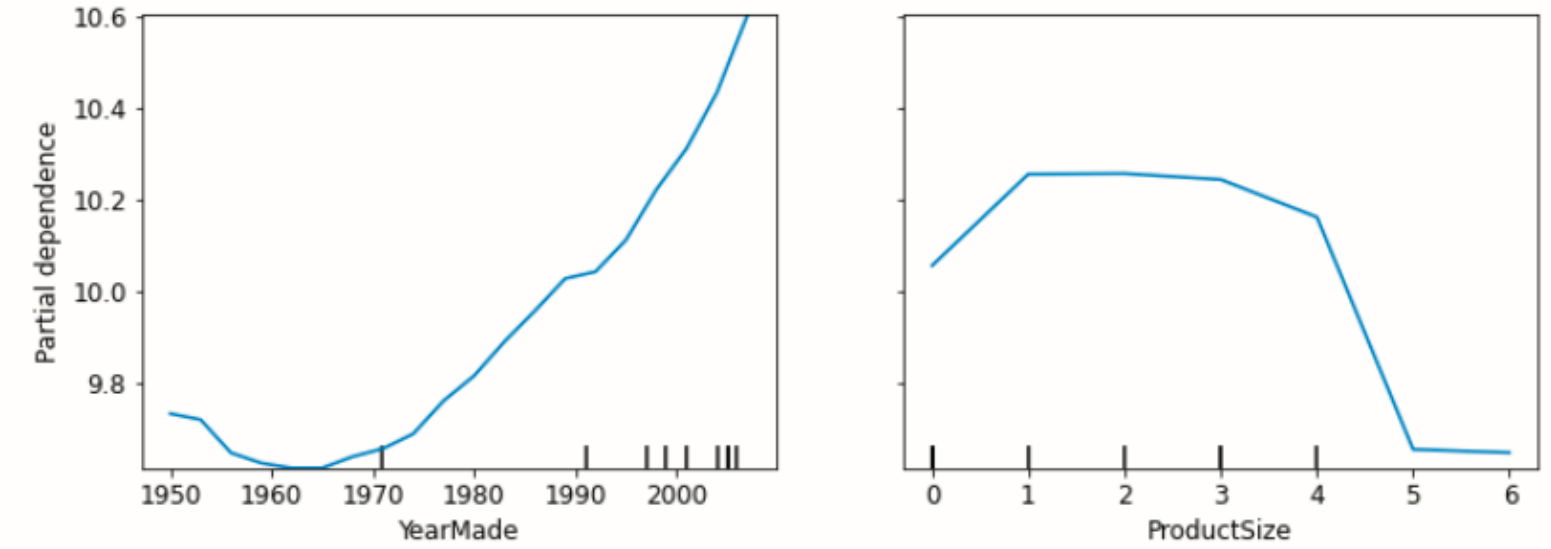
Instead, what we do is replace every single value in the `YearMade` column with 1950, and then calculate the predicted sale price for every auction, and take the average over all auctions. Then we do the same for 1951, 1952, and so forth until our final year of 2011. This isolates the effect of only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

> A: If you are philosophically minded it is somewhat dizzying to contemplate the different kinds of hypotheticality that we are juggling to make this calculation. First, there's the fact that *every* prediction is hypothetical, because we are not noting empirical data. Second, there's the point that we're *not* merely interested in asking how sale price would change if we changed `YearMade` and everything else along with it. Rather, we're very specifically asking, how sale price would change in a hypothetical world where only `YearMade` changed. Phew! It is impressive that we can ask such questions. I recommend Judea Pearl and Dana Mackenzie's recent book on causality, *The Book of Why* (Basic Books), if you're interested in more deeply exploring formalisms for analyzing these subtleties.

With these averages, we can then plot each of these years on the x-axis, and each of the predictions on the y-axis. This, finally, is a partial dependence plot. Let's take a look:

```
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);
```



Looking first of all at the YearMade plot, and specifically at the section covering the years after 1990 (since as we noted this is where we have the most data), we can see a nearly linear relationship between year and price. Remember that our dependent variable is after taking the logarithm, so this means that in practice there is an exponential increase in price. This is what we would expect: depreciation is generally recognized as being a multiplicative factor over time, so, for a given sale date, varying year made ought to show an exponential relationship with sale price.

The ProductSize partial plot is a bit concerning. It shows that the final group, which we saw is for missing values, has the lowest price. To use this insight in practice, we would want to find out why it's missing so often, and what that means. Missing values can sometimes be useful predictors—it entirely depends on what causes them to be missing. Sometimes, however, they can indicate data leakage.

2. Particular Rows and thier Effects

Tree Interpreter

```
!pip install treeinterpreter
```

```
!pip install treeinterpreter
!pip install waterfallcharts
```

```
import warnings
warnings.simplefilter('ignore', FutureWarning)

from treeinterpreter import treeinterpreter
from waterfall_chart import plot as waterfall
```

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.

```
row = valid_xs_final.iloc[:5]
```

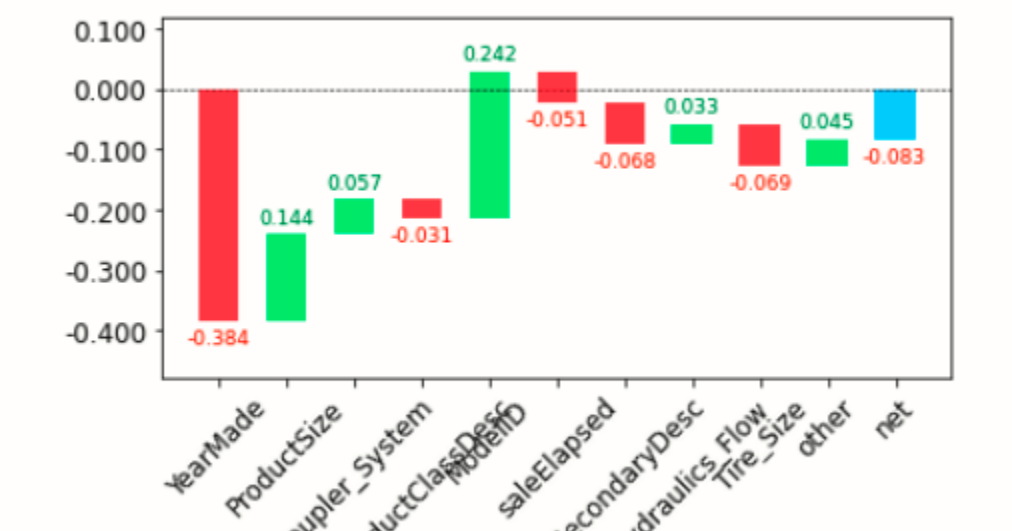
```
prediction,bias,contributions = treeinterpreter.predict(m, row.values)
```

- prediction is simply the prediction that the random forest makes.
- bias is the prediction based on taking the mean of the dependent variable (i.e., the model that is the root of every tree).
- contributions is the most interesting bit—it tells us the total change in predicition due to each of the independent variables. Therefore, the sum of contributions plus bias must equal the prediction, for each row. Let's look just at the first row:

```
prediction[0], bias[0], contributions[0].sum()
```

```
(array([10.02097815]), 10.104140184217215, -0.08316203871169882)
```

```
waterfall(valid_xs_final.columns, contributions[0], threshold=0.08,
          rotation_value=45,formatting='{:,.3f}');
          # how important was each factor
```



Data Leakage

missing value interpretation Check whether the accuracy of the model is too good to be true. Look for important predictors that don't make sense in practice. Look for partial dependence plot results that don't make sense in practice. Thinking back to our bear detector, this mirrors the advice that we provided in <>—it is often a good idea to build a model first and then do your data cleaning, rather than vice versa. The model can help you identify potentially problematic data issues.

It can also help you identify which factors influence specific predictions, with tree interpreters.

The Exploration Problem

A problem with random forests, like all machine learning or deep learning algorithms, is that they don't always generalize well to new data. We will see in which situations neural networks generalize better, but first, let's look at the extrapolation problem that random forests have.

Finding Out-of-Domain Data

ombine our training and validation sets together, create a dependent variable that represents which dataset each row comes from, build a random forest using that data, and get its feature importance:

```
df_dom = pd.concat([xs_final, valid_xs_final])
is_valid = np.array([0]*len(xs_final) + [1]*len(valid_xs_final))

m = rf(df_dom, is_valid)
rf_feat_importance(m, df_dom)[:6]
```

	cols	imp
5	saleElapsed	0.885443
10	SalesID	0.088579
12	MachineID	0.020148
4	ModelID	0.001061
0	YearMade	0.000935
13	Hydraulics	0.000785

Let's get a baseline of the original random forest model's RMSE, then see what the effect is of removing each of these columns in turn:

```
m = rf(xs_final, y)
print('orig', m_rmse(m, valid_xs_final, valid_y))

for c in ('SalesID','saleElapsed','MachineID'):
    m = rf(xs_final.drop(c,axis=1), y)
    print(c, m_rmse(m, valid_xs_final.drop(c,axis=1), valid_y))
```

```
orig 0.233216
SalesID 0.230816
saleElapsed 0.234869
MachineID 0.231053
```

It looks like we should be able to remove SalesID and MachineID without losing any accuracy. Let's check:

```
time_vars = ['SalesID','MachineID']
xs_final_time = xs_final.drop(time_vars, axis=1)
valid_xs_time = valid_xs_final.drop(time_vars, axis=1)

m = rf(xs_final_time, y)
m_rmse(m, valid_xs_time, valid_y)
```

```
0.228506
```

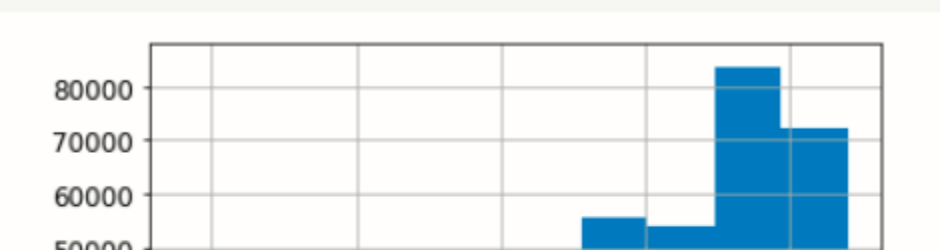
Important Point

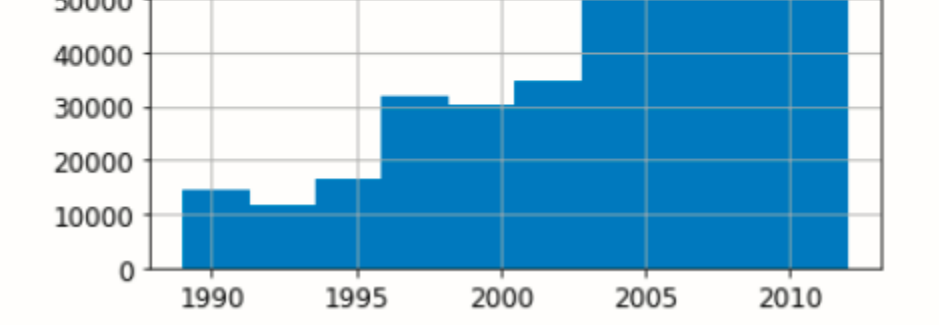
Removing these variables has slightly improved the model's accuracy; but more importantly, it should make it more resilient over time, and easier to maintain and understand. We recommend that for all datasets you try building a model where your dependent variable is is_valid, like we did here. It can often uncover subtle domain shift issues that you may otherwise miss.

Avoid Using Old Data

One thing that might help in our case is to simply avoid using old data. Often, old data shows relationships that just aren't valid any more. Let's try just using the most recent few years of the data:

```
xs['saleYear'].hist();
```






```
filt = xs['saleYear']>2004
xs_filt = xs_final_time[filt]
y_filt = y[filt]
```

```
m = rf(xs_filt, y_filt)
m_rmse(m, xs_filt, y_filt), m_rmse(m, valid_xs_time, valid_y)
```

```
    (0.177674, 0.228469)
```

bit better, which shows that you shouldn't always just use your entire dataset; sometimes a subset can be better.

Using Deep Learning

using NN

```
df_nn = pd.read_csv('TrainAndValid.csv', low_memory=False)
  #ordinal approach
df_nn['ProductSize'] = df_nn['ProductSize'].astype('category')
df_nn['ProductSize'].cat.set_categories(sizes, ordered=True, inplace=True)
 # log of prediccting variable 'Sales' as per condition of competition
df_nn[dep_var] = np.log(df_nn[dep_var])
 # date part
df_nn = add_datepart(df_nn, 'saledate')
```

trim unwanted columns in the random forest by using the same set of columns for our neural network

```
df_nn_final = df_nn[list(xs_final_time.columns) + [dep_var]]
```

Categorial Variables

Using Embeddings

- Categorical columns are handled very differently in neural networks, compared to decision tree approaches. As we saw in <>, in a neural net a great way to handle categorical variables is by using embeddings. To create embeddings, fastai needs to determine which columns should be treated as categorical variables.

categorical variables.

- **It does this by comparing the number of distinct levels in the variable to the value of the max_card parameter. If it's lower, fastai will treat the variable as categorical.**
- Embedding sizes larger than 10,000 should generally only be used after you've tested whether there are better ways to group the variable, so we'll use 9,000 as our max_card:

```
cont_nn,cat_nn = cont_cat_split(df_nn_final, max_card=9000, dep_var=dep_var)
```

In this case, there's one variable that we absolutely do not want to treat as categorical: the saleElapsed variable. A categorical variable cannot, by definition, extrapolate outside the range of values that it has seen, but we want to be able to predict auction sale prices in the future. Let's verify that cont_cat_split did the correct thing.

```
cont_nn
```

```
['saleElapsed']
```

cardinality of each of the categorical variables that we have chosen so fa

```
df_nn_final[cat_nn].nunique()
```

```
YearMade                73
ProductSize              6
Coupler_System           2
fiProductClassDesc      74
ModelID               5281
fiSecondaryDesc        177
Hydraulics_Flow          3
fiModelDesc           5059
Enclosure                6
fiModelDescriptor      140
Hydraulics              12
Tire_Size               17
ProductGroup             6
Drive_System             4
dtype: int64
```

Removing Redundency

```
xs_filt2 = xs_filt.drop('fiModelDescriptor', axis=1)
valid_xs_time2 = valid_xs_time.drop('fiModelDescriptor', axis=1)
m2 = rf(xs_filt2, y_filt)
m_rmse(m2, xs_filt2, y_filt), m_rmse(m2, valid_xs_time2, valid_y)
```

```
(0.176777, 0.230648)
```

very low impact so remove

```
cat_nn.remove('fiModelDescriptor')
```

create our TabularPandas object in the same way as when we created our random forest, with one very important addition: normalization.

```
procs_nn = [Categorify, FillMissing, Normalize]
to_nn = TabularPandas(df_nn_final, procs_nn, cat_nn, cont_nn,
                      splits=splits, y_names=dep_var)
```

tabular require less GPU, so use bigger batch size

```
dls = to_nn.dataloaders(1024)
```

Setting y-range for regression models

```
y = to_nn.train.y
y.min(),y.max()
```

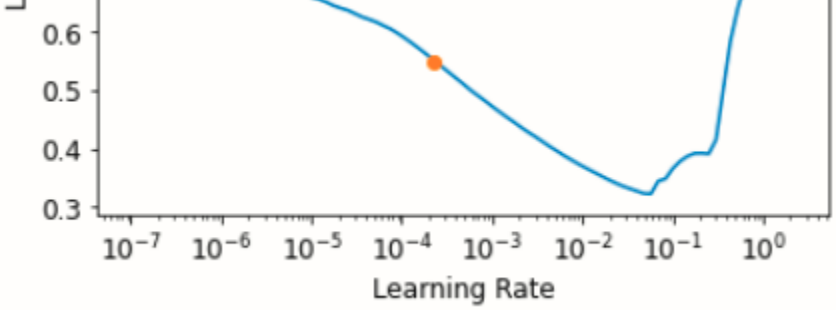
```
(8.465899, 11.863583)
```

By default, for tabular data fastai creates a neural network with two hidden layers, with 200 and 100 activations, respectively. This works quite well for small datasets, but here we've got quite a large dataset, so we increase the layer sizes to 500 and 250:

```
learn = tabular_learner(dls, y_range=(8,12), layers=[500,250],
                        n_out=1, loss_func=F.mse_loss)
```

Finding Good Learning Rate

```
learn.lr_find()
```




```
learn.fit_one_cycle(5, 1e-2)
```

epoch	train_loss	valid_loss	time
0	0.048319	0.056736	00:40
1	0.046619	0.069091	00:41
2	0.043961	0.051723	00:40
3	0.040681	0.053780	00:39
4	0.038691	0.051607	00:41

Comparing Results

- Random Forest
- Neural Network

```
preds,targs = learn.get_preds()
r_mse(preds,targs)
```

```
0.227172
```

```
learn.save('nn')
```

```
Path('models/nn.pth')
```

Ensembling

- average both **Random Forest** and **NN** their predictions would be better than either one's individual predictions.

Approach

So far our approach to ensembling has been to use bagging, which involves combining many models (each trained on a different data subset) together by averaging them

(each trained on a different data subset) together by averaging them.

```
rf_preds = m.predict(valid_xs_time)
ens_preds = (to_np(preds.squeeze()) + rf_preds) /2
```

```
r_mse(ens_preds,valid_y)
```

```
0.222104
```

This gives us a better result than either model achieved on its own

Conclusion

Conclusion: Our Advice for Tabular Modeling

We have dicussed two approaches to tabular modeling: decision tree ensembles and neural networks. We've also mentioned two different decision tree ensembles: random forests, and gradient boosting machines. Each is very effective, but each also has compromises:

- *Random forests* are the easiest to train, because they are extremely resilient to hyperparameter choices and require very little preprocessing. They are very fast to train, and should not overfit if you have enough trees. But they can be a little less accurate, especially if extrapolation is required, such as predicting future time periods.
- *Gradient boosting machines* in theory are just as fast to train as random forests, but in practice you will have to try lots of different hyperparameters. They can overfit, but they are often a little more accurate than random forests.
- *Neural networks* take the longest time to train, and require extra preprocessing, such as normalization; this normalization needs to be used at inference time as well. They can provide great results and extrapolate well, but only if you are careful with your hyperparameters and take care to avoid overfitting.

We suggest starting your analysis with a random forest. This will give you a strong baseline, and you can be confident that it's a reasonable starting point. You can then use that model for feature selection and partial dependence analysis, to get a better understanding of your data.

From that foundation, you can try neural nets and GBMs, and if they give you significantly better results on your validation set in a reasonable amount of time, you can use them. If decision tree ensembles are working well for you, try adding the embeddings for the categorical variables to the data, and see if that helps your decision trees learn better.