Tabular

In [ ]:
```
from fastbook import *
# from kaggle import api
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
```

In [ ]:
```
dls = TabularDataLoaders.from_csv(path/'adult.csv', path=path, y_names="salary",
    cat_names = ['workclass', 'education', 'marital-status', 'occupation',
                 'relationship', 'race'],
    cont_names = ['age', 'fnlwgt', 'education-num'],
    procs = [Categorify, FillMissing, Normalize])
learn = tabular_learner(dls, metrics=accuracy)
```

There is **no pretrained model available** for this task (in general, pretrained models are not widely available for any tabular modeling tasks, although some organizations have created them for internal use), so we **don't use fine_tune** in this case. Instead we **use fit_one_cycle, the most commonly used method for training fastai models from scratch (i.e. without transfer learning)**b

In [ ]:
```
learn.fit_one_cycle(3)
# epoch train_loss      valid_loss      accuracy        time
# 0     0.386528        0.368881        0.828778        00:06
# 1     0.369635        0.346222        0.840756        00:06
# 2     0.357984        0.344679        0.841216        00:06
```

Recent studies have shown that the vast majority of datasets can be best modeled with just **two methods:**

1. **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)
2. **Multilayered neural networks** learned with SGD (i.e., shallow and/or deep learning), mainly for **unstructured data** (such as audio, images, and natural language)**

In [ ]:
```
```

Using TabularPandas and TabularProc

TabularPandas (set of operations) --> TabularProcs (names of those operations) --> Categorify/FillMissing (type of TabularProcs)

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

use `np.where` **, a useful function that returns (as the first element of a tuple) the indices of all** `True` **values:**

In [ ]:
```
cond = (df.saleYear<2011) | (df.saleMonth<10)
train_idx = np.where( cond)[0] # is numpy array
valid_idx = np.where(~cond)[0]
```

```
# so that we predict future ke values (jo condition ke opposite hai )
splits = (list(train_idx),list(valid_idx))
```

In [ ]:
```
# TabularPandas needs to be told which columns are continuous and which are
# categorical. We can handle that automatically using the helper function cont_cat_split:
dep_var = 'SalePrice'
cont,cat = cont_cat_split(df, 1, dep_var=dep_var) # 1??
to = TabularPandas(df, procs, cat, cont, y_names=dep_var, splits=splits)
```

A `TabularPandas` behaves a lot like a fastai `Datasets` object, including providing `train` and `valid` attributes:

In [ ]:
```
# ye sab features milte hai
to.train, to.valid
to.show(3) # will *show* categorical as categorical, but they are stored as numeric
to.items # will have everyhing converted to numeric items
# 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.
# except if ordered category column hai, to that order me number milenge
```

In [ ]:
```
xs,y = to.train.xs,to.train.y
valid_xs,valid_y = to.valid.xs,to.valid.y
```

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

In [ ]:
```
m = DecisionTreeRegressor()
# can add (example values)
# min_samples_leaf (=25), max_leaf_nodes (=4)
m.fit(xs, y)
```

In [ ]:
```

```

viewing a decision tree

In [ ]:
```
draw_tree(m, xs, size=10, leaves_parallel=True, precision=2)
# or
samp_idx = np.random.permutation(len(y))[:500]
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')
```

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!

In [ ]:

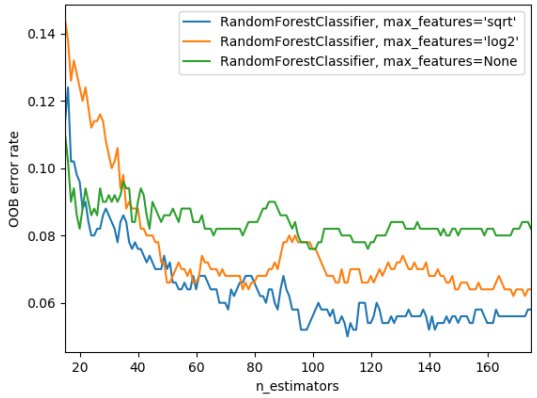
random forest

use bagging on decision trees

create random forest just like a decision tree, and also specifying parameters

- **n_estimators** defines the number of trees we want. set 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** defines how many rows to sample for training each tree. 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** defines how many columns to sample at each split point (where 0.5 means "take half the total number of columns") 0.5 is nice value
- **min_samples_leaf** for when to stop splitting the tree nodes, effectively limiting the depth of the tree parameter we used in the last section. nice value 4
- **n_jobs=-1** to tell sklearn to use all our CPUs to build the trees in parallel.

The sklearn docs show an example of the effects of different `max_features` choices, with increasing numbers of trees. In the plot, the blue plot line uses the fewest features and the green line uses the most (it uses all the features). As you can see, **the models with the lowest error result from using a subset of features but with a larger number of trees.**



In [ ]:

```
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_features=max_features,
        min_samples_leaf=min_samples_leaf, oob_score=True).fit(xs, y)
```

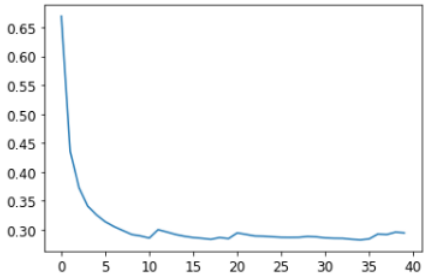
In [ ]:

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

what happens to the RMSE as we add more and more trees.

In [ ]:

```
plt.plot([r_mse(preds[:i+1].mean(0), valid_y) for i in range(40)]);
```



overfitting after 40 trees, we cannot say for now

In [ ]:

```
```

Out-of-Bag Error

1.0 for a perfect model and 0.0 for a random model

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.

This is particularly beneficial in cases where we have only a small amount of training data

In [ ]:

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

In [ ]:

```
```

Tree Variance for Prediction Confidence

model averages the individual tree's predictions to get an overall prediction (an estimate of the value). confidence of the estimate by use the standard deviation of predictions across the trees, instead of just the mean. 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).

In [ ]:

```
preds = np.stack([t.predict(valid_xs) for t in m.estimators_])
preds_std = preds.std(0)
preds_std[:5] # array([0.33425027, 0.25110331, 0.08780472, 0.21735859, 0.29817466])
# if using this model to decide what items to bid on at auction, a low confdence
# prediction might cause you to look more carefully at an item before you made a bid.
```

In [ ]:

```
```

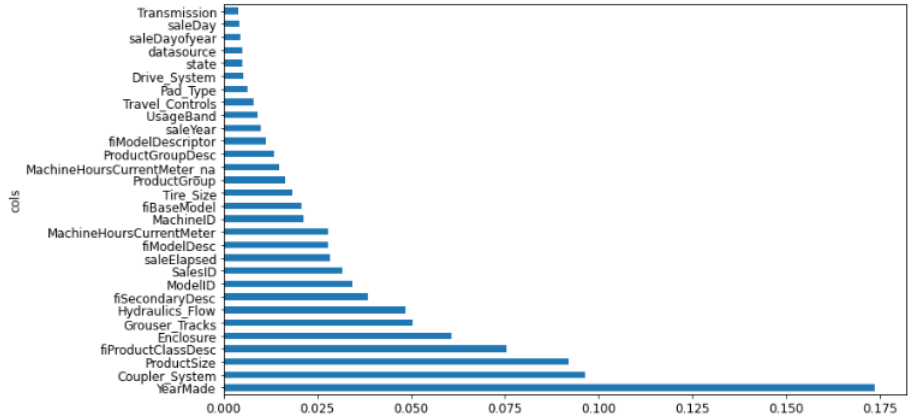
Feature Importance

In [ ]:

```
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[:5]
#       cols    imp
# 56    YearMade        0.173521
```

```
# 30    Coupler_System  0.096459
# 6     ProductSize     0.091934
```

In [ ]:
```
def plot_fi(fi):
    return fi.plot('cols', 'imp', 'barh', figsize=(12,7), legend=False)
plot_fi(fi[:30]);
```



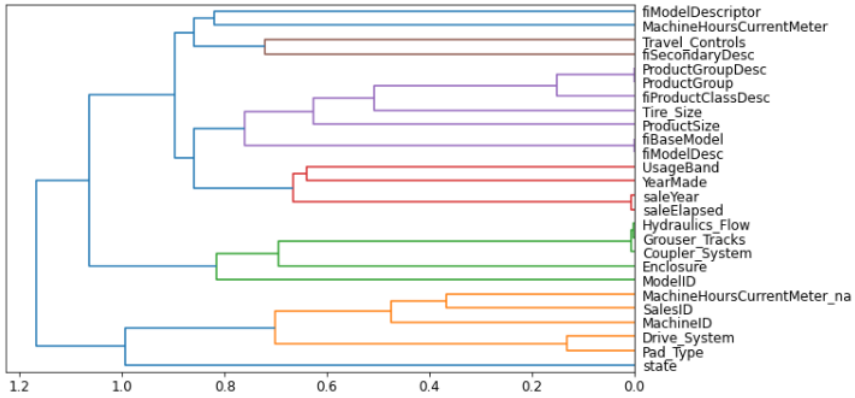
remove low-imp variables

In [ ]:
```
to_keep = fi[fi.imp>0.005].cols
# then retrain
xs_imp = xs[to_keep]
valid_xs_imp = valid_xs[to_keep]
m = rf(xs_imp, y)
```

In [ ]:
```
```

remove redundant features

In [ ]:
```
cluster_columns(xs_imp)
```



quikcly trains a random forest and returns the OOB score, by using a lower `max_samples` and higher `min_samples_leaf`

In [ ]:
```
def get_oob(df):
    m = RandomForestRegressor(n_estimators=40, min_samples_leaf=15,
        max_samples=5000, max_features=0.5, n_jobs=-1, oob_score=True)
    m.fit(df, y)
    return m.oob_score_
baseline_score = get_oob(xs_imp)
```

In [ ]:
```
# remove each of theseone by one
result = {c:get_oob(xs_imp.drop(c, axis=1)) for c in [
    'saleYear', 'saleElapsed', 'ProductGroupDesc','ProductGroup',
    'fiModelDesc', 'fiBaseModel',
    'Hydraulics_Flow','Grouser_Tracks', 'Coupler_System']
# ({'saleYear': 0.8441887602263386, 'saleElapsed': 0.839770948235284, ...})
```

In [ ]:
```
# drop these together
to_drop = ['saleYear', 'ProductGroupDesc', 'fiBaseModel', 'Grouser_Tracks']
get_oob(xs_imp.drop(to_drop, axis=1))
```

In [ ]:
```
xs_final = xs_imp.drop(to_drop, axis=1)
valid_xs_final = valid_xs_imp.drop(to_drop, axis=1)
```

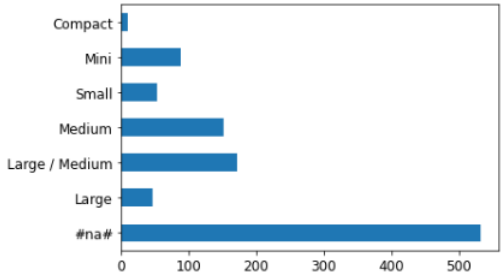
In [ ]:
```

```

count of values per category & partial dependence

In [ ]:
```
p = valid_xs_final['ProductSize'].value_counts(sort=False).plot.barh()
c = to.classes['ProductSize']
```

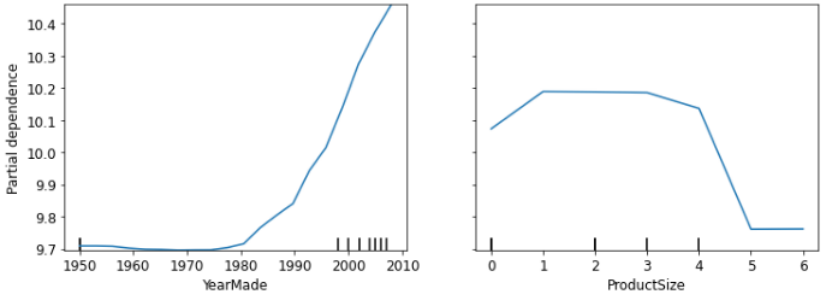
```
plt.yticks(range(len(c)), c);
```



In [ ]:
```
valid_xs_final['YearMade'].hist() # isko bhi kar sakte
```

In [ ]:
```
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);
```



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 --> Data Leakage

is partial dependence plots do not make sense in real life, then Data Leakage

Tree interpreter

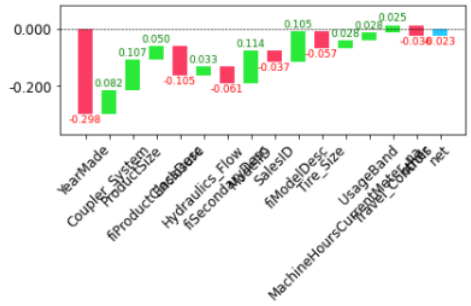
In [ ]:
```
import warnings
warnings.simplefilter('ignore', FutureWarning)
from treeinterpreter import treeinterpreter
from waterfall_chart import plot as waterfall
```

In [ ]:
```
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:**

In [ ]:
```
waterfall(valid_xs_final.columns, contributions[0], threshold=0.08,
          rotation_value=45,formatting='{:,.3f}');
```

shows how the positive and negative contributions from all the independent variables sum up to create the final prediction, which is the righthand column labeled "net" here:



most useful in production, rather than during model development.

In [ ]:

In [ ]:

rf cannot generalise on out-of-domain data

That's why we need to make sure our validation set does not contain out-of-domain data.

In [ ]:

Finding Out-of-Domain Data

basically they are causing problems, training and validation me wo alag alag behave karte hai

use rf only! predict whether a row is in the validation set or the training set.

In [ ]:
```
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
# 9     saleElapsed     0.689527
# 8     SalesID 0.207635
# 12    MachineID       0.098930
```

three columns that differ significantly between the training and validation sets: saleElapsed, SalesID, and MachineID. It's fairly obvious why for saleElapsed: it's the number of days between the start of the dataset and each row, so it directly encodes the date. (remember validation set was mostly later dates sice we want to predict the future) The difference in SalesID suggests that identifiers for auction sales might increment over time. MachineID suggests something similar might be happening for individual items sold in those auctions.

how to drop these

In [ ]:
```
# individual drop karna
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 0.296279
# SalesID 0.264977
# saleElapsed 0.302022
# MachineID 0.297024
```

In [ ]:
```
# should be able to remove SalesID and MachineID without losing any accuracy
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)
# 0.26072
# slightly improved the model's accuracy; but more importantly,
# it should make it more resilient over time
```

In [ ]:
```

```

Use Neural Network

In [ ]:
```
#  Let's first replicate the steps we took to set up the TabularPandas object
df_nn = pd.read_csv('Tabular_Data/TrainAndValid.csv', low_memory=False)
df_nn['ProductSize'] = df_nn['ProductSize'].astype('category')
df_nn['ProductSize'].cat.set_categories(sizes, ordered=True, inplace=True)
df_nn[dep_var] = np.log(df_nn[dep_var])
df_nn = add_datepart(df_nn, 'saledate')
```

In [ ]:
```
this_list = list(xs_final_time.columns) + [dep_var]
df_nn = df_nn.reindex(columns=this_list) # this is (empty?) df with column name rovided
df_nn_final = df_nn[this_list] # add the cols
#
```

Categorical columns are handled differently in nns , here use embeddings. To create embeddings, fastai needs to determine which columns should be treated as categorical variables. It does this by comparing the number of distinct levels in the variable to the value of the `max_card` parameter. (card stands for cardinality maybe) 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` :

In [ ]:
```
cont_nn,cat_nn = cont_cat_split(df_nn_final, max_card=9000, dep_var=dep_var)
# these are just lists of columns
# to remove anything, just use list operations
```

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.

very important addition: normalization.

In [ ]:
```
procs_nn = [Categorify, FillMissing, Normalize]
to_nn = TabularPandas(df_nn_final, procs_nn, cat_nn, cont_nn,
                      splits=splits, y_names=dep_var)
# Tabular models and data don't generally require much GPU RAM,
# so we can use larger batch sizes:
dls = to_nn.dataloaders(1024)
```

good idea to set y_range for regression models

In [ ]:
```
y = to_nn.train.y
y.min(),y.max()
# (8.465899, 11.863583)
```

In [ ]:
```
learn = tabular_learner(dls, y_range=(8,12), layers=[500,250],
                        n_out=1, loss_func=F.mse_loss)
# two layers of sizes 500, 250 use kiya hai
learn.lr_find()
learn.fit_one_cycle(5, 1e-2)
# epoch train_loss      valid_loss      time
# 0     0.204266        0.791131        00:00...
# 4     0.043403        0.062707        00:00
```

In [ ]:
```
preds,targs = learn.get_preds()
r_mse(preds,targs)
```

In [ ]:
```

```

In [ ]:
```

```

Ensembling

two very different models trained using very different algorithms: a random forest, and a neural network. It would be reasonable to expect that the kinds of errors that each one makes would be quite different. Therefore the average of their predictions would be better than either one's individual predictions.

In [ ]:
```
rf_preds = m.predict(valid_xs_time) # gives rank 1 array
ens_preds = (to_np(preds.squeeze()) + rf_preds) /2 # since preds is tensor
```

- **.*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.

In [ ]:
```
```