In [1]:

```
!pip install -Uqq fastbook
!pip install -Uqq fastbook kaggle waterfallcharts treeinterpreter dtreeviz
import fastbook
fastbook.setup_book()
from fastbook import *
from fastai.vision.widgets import *
from google.colab import drive
from pandas.api.types import is_string_dtype, is_numeric_dtype, is_categorical_d
type
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
drive.mount('/content/drive')
```

```
     |████████████████████████████████| 727kB 16.2MB/s
     |████████████████████████████████| 1.1MB 23.5MB/s
     |████████████████████████████████| 194kB 52.9MB/s
     |████████████████████████████████| 51kB 8.9MB/s
     |████████████████████████████████| 61kB 9.4MB/s
     |████████████████████████████████| 51kB 6.4MB/s
     |████████████████████████████████| 204.2MB 80kB/s
     |████████████████████████████████| 204kB 59.0MB/s
  Building wheel for waterfallcharts (setup.py) ... done
  Building wheel for dtreeviz (setup.py) ... done
  Building wheel for pyspark (setup.py) ... done
Mounted at /content/gdrive
Mounted at /content/drive
```

In [2]:

```
!pip install kaggle

!mkdir ~/.kaggle
!echo '{"username":"","key":""}' > ~/.kaggle/kaggle.json
!chmod 600 /root/.kaggle/kaggle.json


from kaggle import api
```

```
Requirement already satisfied: kaggle in /usr/local/lib/python3.6/di
st-packages (1.5.10)
Requirement already satisfied: python-slugify in /usr/local/lib/pyth
on3.6/dist-packages (from kaggle) (4.0.1)
Requirement already satisfied: certifi in /usr/local/lib/python3.6/d
ist-packages (from kaggle) (2020.12.5)
Requirement already satisfied: python-dateutil in /usr/local/lib/pyt
hon3.6/dist-packages (from kaggle) (2.8.1)
Requirement already satisfied: requests in /usr/local/lib/python3.6/
dist-packages (from kaggle) (2.23.0)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.6/d
ist-packages (from kaggle) (1.24.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.6/dist
-packages (from kaggle) (4.41.1)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.
6/dist-packages (from kaggle) (1.15.0)
Requirement already satisfied: text-unidecode>=1.3 in /usr/local/li
b/python3.6/dist-packages (from python-slugify->kaggle) (1.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python
3.6/dist-packages (from requests->kaggle) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/p
ython3.6/dist-packages (from requests->kaggle) (3.0.4)
```

In [3]:

```
path = URLs.path('bluebook')
path
Path.BASE_PATH = path

if not path.exists():
    path.mkdir(parents=true)
    api.competition_download_cli('bluebook-for-bulldozers', path=path)
    file_extract(path/'bluebook-for-bulldozers.zip')
path.ls(file_type='text')
```

```
 19%|██         | 9.00M/48.4M [00:00<00:00, 57.5MB/s]

Downloading bluebook-for-bulldozers.zip to /root/.fastai/archive/blu
ebook

100%|██████████| 48.4M/48.4M [00:00<00:00, 144MB/s]
```

Out[3]:

```
(#7) [Path('random_forest_benchmark_test.csv'),Path('Valid.csv'),Pat
h('median_benchmark.csv'),Path('Machine_Appendix.csv'),Path('ValidSo
lution.csv'),Path('TrainAndValid.csv'),Path('Test.csv')]
```

In [4]:

```
df = pd.read_csv(path/'TrainAndValid.csv', low_memory=False)
df = add_datepart(df, 'saledate') #adds data such as year of sale , month of sal
e etc relating to dates

df_test = pd.read_csv(path/'Test.csv', low_memory=False)
df_test = add_datepart(df_test, 'saledate')#adds data such as year of sale , mon
th of sale etc relating to dates

x = [o for o in df.columns if o.startswith('sale')]
# x #Shows all the new columns that were added , information that was extracted
 via add_datepart
```

In [5]:

```
procs = [FillMissing,Categorify]#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 valu
e was missing.

cond = ((df.saleYear<=2010) | ((df.saleYear==2011) & (df.saleMonth<=10)))

train_idx = np.where( cond)[0]
valid_idx = np.where(~cond)[0]
#we will let validation set be dates after  November 2011
splits = (list(train_idx),list(valid_idx))

pd.options.display.max_rows = 20
pd.options.display.max_columns = 10

df[["saleYear","saleMonth"]]

dep_var = 'SalePrice'#dependant variable
df[dep_var] = np.log(df[dep_var])


cont,cat = cont_cat_split(df, 1, dep_var=dep_var)#split our columns into  contin
uous data columns (salesID,yearmade etc) and categorical

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

In [6]:

```
len(tt.cont_names) #names of the continous variables
len(tt.cat_names) == len(tt.classes) #classes only include categorical data
```

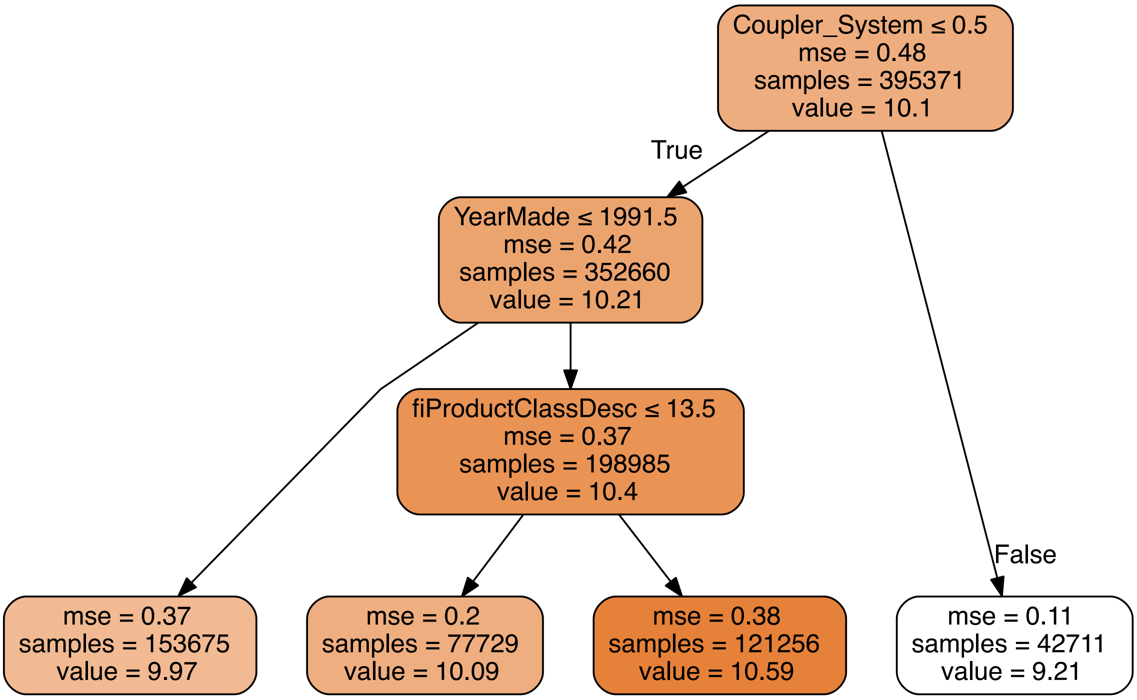
Out[6]:

```
True
```

In [7]:

```
save_pickle(path/'tt.pkl',tt)
tt = load_pickle(path/'tt.pkl')

xs,y = tt.train.xs,tt.train.y
valid_xs,valid_y = tt.valid.xs,tt.valid.y

m = DecisionTreeRegressor(max_leaf_nodes=4)#draw
m.fit(xs, y);

draw_tree(m, xs, size=11, leaves_parallel=True, precision=2)
```

Out[7]:



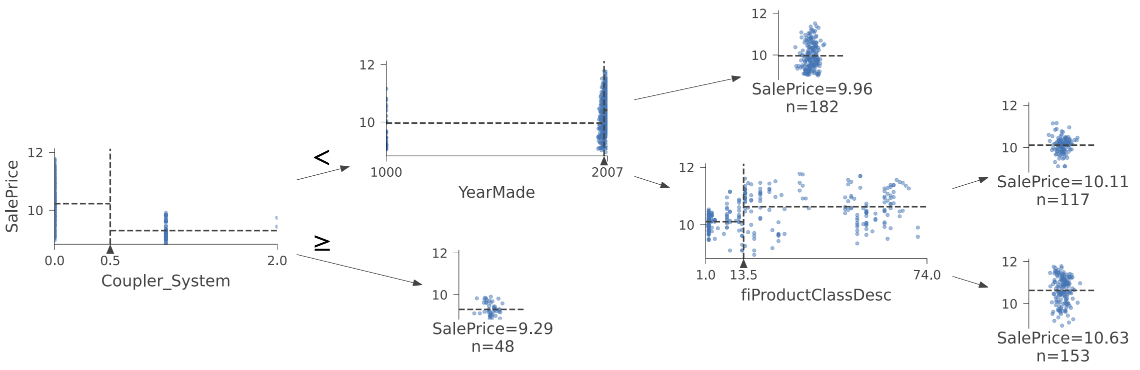
In [7]:

```

```

In [8]:

```
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')
```

Out[8]:



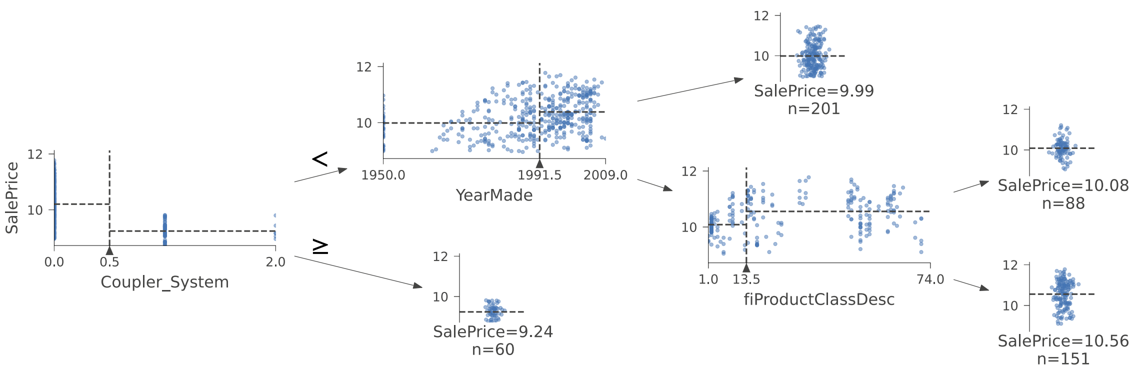
In [9]:

```
#As seen above , in the YearMade Split we can see that ther are bulldozers made
 in 1000 ... that's just wrong,
#this may be due to missing values so lets replace any date before 1900 with an
 aribtrary year , 1950
xs.loc[xs['YearMade']<1900, 'YearMade'] = 1950
valid_xs.loc[valid_xs['YearMade']<1900, 'YearMade'] = 1950
# df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
#                    columns=['a', 'b', 'c'])


# #for renaming values , lets say change all values in column "a" which are less
than 3 to"less than 3"

# df2.loc[df2['a']<3 , 'a'] = "less than 3"
# df2
```

In [10]:

```
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')
```

Out[10]:



The image above allows us to look into our data better

In [11]:

```
m = DecisionTreeRegressor()
m.fit(xs, y);#let's make a decision tree  with no limit on the max number of lea
f nodes
```

In [12]:

```
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) #predict on data (xs) and c
ompare with label (y)
```

In [13]:

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

Out[13]:

```
1e-06
```

In [14]:

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

Out[14]:

```
0.366669
```

In [15]:

```
m.get_n_leaves(), len(xs)
```

Out[15]:

```
(317382, 395371)
```

In [16]:

```
m = DecisionTreeRegressor(min_samples_leaf=25)
m.fit(xs, y)
m_rmse(m, xs, y), m_rmse(m, valid_xs, valid_y)
```

Out[16]:

```
(0.212222, 0.283671)
```

In [17]:

```
m.get_n_leaves(), len(xs)
```

Out[17]:

```
(12117, 395371)
```

In [18]:

```
#Let's create our Random Forest

def rf(xs, y, n_estimators=40, max_samples=200_000,
       max_features=0.5, min_samples_leaf=5, **kwargs):#xs and y are our depende
nt and dependent variables respectively ,
       #n_estimators tells us to generate 40 trees
       #max_samples tells us how many randomly chosen rows(horizontal) to sample
for each tree if we have more than 200,000 datapoints we set it to equal to 200,
000
       #if not we let it revert to default value
      #min_samples_leaf sets the minimum number of samples(data points)  to be a
t each node
    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)
```

In [19]:

```
random_forest_classifier = rf(xs, y);
```

In [20]:

```
m_rmse(random_forest_classifier, xs, y), m_rmse(random_forest_classifier, valid_
xs, valid_y)#drastic improvement from using the standard
#decision tree
```

Out[20]:

```
(0.170534, 0.244362)
```

In [21]:

```
preds = np.stack([t.predict(valid_xs) for t in random_forest_classifier.estimato
rs_])

pd.DataFrame(data=preds )
#the 0 axis (horizontals) shows the predicitions given by each of the 40 random_
forest trees we generated , for each datapoint in
#valid_xs(prediction made by each random_forest tree on valid_xs)
```

Out[21]:

	0	1	2	3	4	...	17322	17323	17324
0	10.531863	10.021294	9.498024	11.372683	9.924388	...	9.557606	9.203068	9.557606
1	10.549086	9.738327	9.382929	10.926208	10.363726	...	9.132804	9.650739	9.132804
2	10.382526	9.617176	9.341506	10.380203	10.414758	...	9.246454	9.770592	9.186619
3	10.575123	9.979257	9.541314	10.790382	10.609635	...	9.364809	10.140739	9.364809
4	11.118372	9.909673	9.188512	10.821608	10.655706	...	9.366326	9.743862	9.617640
...	...	...	...	...	...	...	...	...	...
35	11.394524	9.605946	9.954809	11.124083	10.612848	...	9.138899	9.173776	9.088428
36	10.620919	10.566007	9.304760	10.696635	10.689721	...	9.128352	9.306363	9.128352
37	10.592668	9.885385	9.705202	10.846403	10.528579	...	9.326178	9.529953	9.402532
38	10.559494	9.964017	9.296854	10.954026	10.572043	...	9.236747	9.236747	9.236747
39	10.547865	9.931116	9.344061	10.895707	10.861056	...	9.632012	9.201807	9.632012

40 rows × 17327 columns

In [22]:

```
# a = np.array([[1, 2], [3, 4]])
# a.mean(0)
```

In [23]:

```
preds.mean(0) #lets get the mean of  each column (vertical  )
#this gives us the average price predicition (across all our 40 trees) for each
 of the 17327 auctions in out validation set
```

Out[23]:

```
array([10.73807379,  9.99333801,  9.60171244, ...,  9.26781559,  9.4
3311306,  9.66437958])
```

In [24]:

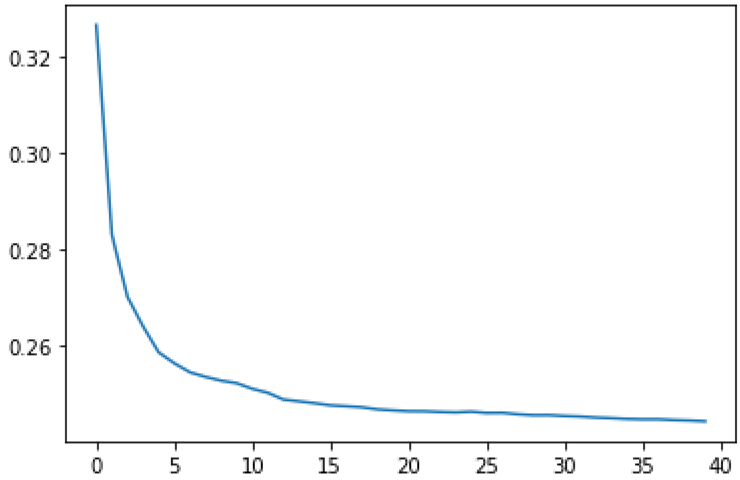
```
r_mse(preds.mean(0), valid_y)
#SEE , SAME AS m_rmse(random_forest_classifier, valid_xs, valid_y)
```

Out[24]:

```
0.244362
```

In [25]:

```
plt.plot([r_mse(preds[:i+1].mean(0), valid_y) for i in range(40)]);
#Treat for the eyes let's see how rmse improves with more trees being added
```



In [26]:

```
pd.DataFrame(data=preds )
#the rows (horzontals) show the predicitions given by each of the 40 random_fore
st trees we generated , for each datapoint in
#valid_xs(prediction made by each random_forest tree on valid_xs)
```

Out[26]:

	0	1	2	3	4	...	17322	17323	17324
0	10.531863	10.021294	9.498024	11.372683	9.924388	...	9.557606	9.203068	9.557606
1	10.549086	9.738327	9.382929	10.926208	10.363726	...	9.132804	9.650739	9.132804
2	10.382526	9.617176	9.341506	10.380203	10.414758	...	9.246454	9.770592	9.186619
3	10.575123	9.979257	9.541314	10.790382	10.609635	...	9.364809	10.140739	9.364809
4	11.118372	9.909673	9.188512	10.821608	10.655706	...	9.366326	9.743862	9.617640
...	...	...	...	...	...	...	...	...	...
35	11.394524	9.605946	9.954809	11.124083	10.612848	...	9.138899	9.173776	9.088428
36	10.620919	10.566007	9.304760	10.696635	10.689721	...	9.128352	9.306363	9.128352
37	10.592668	9.885385	9.705202	10.846403	10.528579	...	9.326178	9.529953	9.402532
38	10.559494	9.964017	9.296854	10.954026	10.572043	...	9.236747	9.236747	9.236747
39	10.547865	9.931116	9.344061	10.895707	10.861056	...	9.632012	9.201807	9.632012

40 rows × 17327 columns

In [27]:

```
preds_std = preds.std(0)#lets get the std of each predicitons made on the same d
atapoint by each our 40 treees

# a = np.array([[1, 2], [3, 4]])
# a.mean(0)# a = np.array([[1, 2], [3, 4]])
# a.mean(0)
```

In [28]:

```
preds_std[:5]#As seen, variance fluctuates quite abit this is because on some au
ctions the trees agree but for the one with high
#std the trees don't quite agree. As such we could say that the confidence in th
e predcitions is rather varied
```

Out[28]:

```
array([0.32064182, 0.23362305, 0.29160316, 0.22396801, 0.27600683])
```

In [29]:

```
r_mse(random_forest_classifier.oob_prediction_, y)#Random Forest uses the baggin
g approach, thus there are subsets of our training dataset
#that would not have been used for training the tree as such we can now validate
our trees against the subsets that
#it wasnt trained on. The error we attain from this is called Out Of Bag Error

#note that comparisons are made to the training labels (y) instead of valid_y si
nce the subsets we are prediciting on come
#from the training dataset
```

Out[29]:

```
0.211113
```

In [30]:

```
def rf_feat_importance(random_forest_classifier, df):
    return pd.DataFrame({'cols':df.columns, 'imp':random_forest_classifier.featu
re_importances_}
                       ).sort_values('imp', ascending=False)

#A function that can give features of our dataset that played the most part in a
ffecting the price of the bulldozer
```

In [31]:

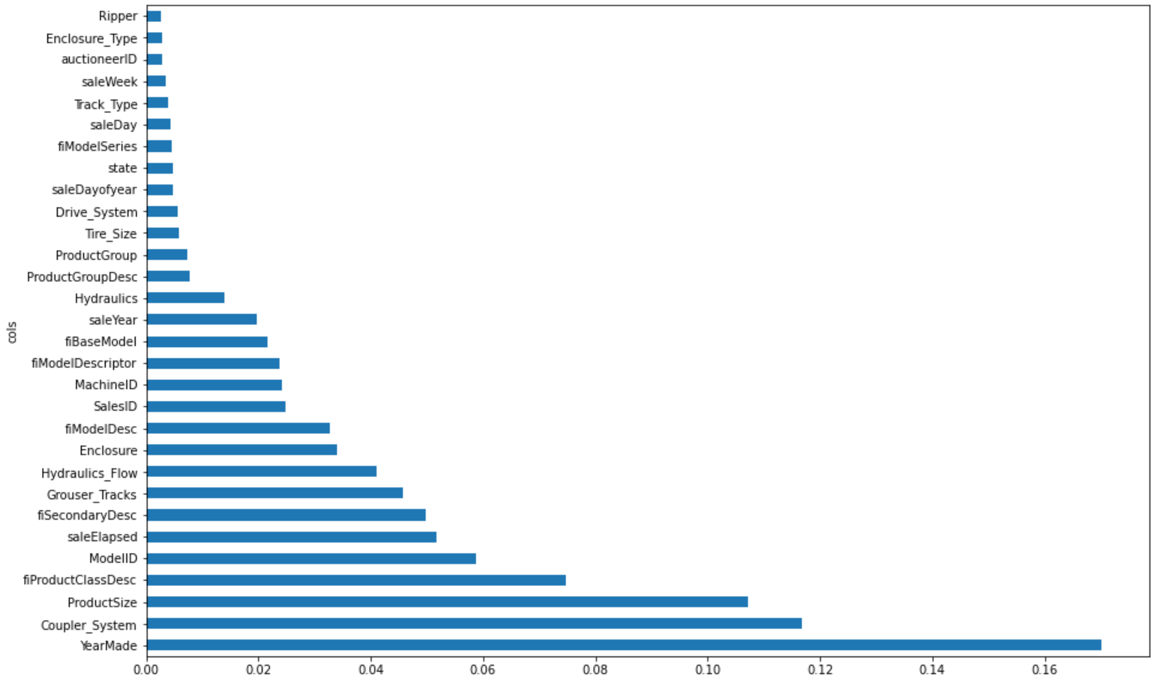
```
fi = rf_feat_importance(random_forest_classifier, xs)
fi[:10]

#What's going on ?

#We start at the top of our first treee and trickle down, at each split we see w
hich column was used to split the data
#we compare the model's prediction before and after the split and relate the Inc
rease or Decrease in the price to the column that
#was used as the binary splitter. We do this to all our 40 trees and add  the Im
provement/Disimprovement for all the columns in all the trees , normalize them
#so they sum up to one and we get the feature importance as shown below.

fi.sum(0)["imp"] #sum of all the values in the horizontals of the column "imp"
#Proven to be essentially 1
```

Out[31]:

```
0.9999999999999999
```

In [32]:

```
def plot_fi(fi):
    return fi.plot('cols', 'imp', 'barh', figsize=(15,10), legend=False)

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



In [33]:

```
#Could we imporve our Random Forest Classifier by removing the variables of low
 importance ?
```

In [34]:

```
to_keep = fi[fi["imp"]>0.005].cols#only keep columns with importance of greater
 than 0.005
```

In [35]:

```
to_keep
```

Out[35]:

```
59                YearMade
31           Coupler_System
7              ProductSize
8       fiProductClassDesc
56                 ModelID
               ...
24              Hydraulics
11        ProductGroupDesc
10            ProductGroup
29               Tire_Size
12            Drive_System
Name: cols, Length: 21, dtype: object
```

In [36]:

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

In [37]:

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

In [38]:

```
m_rmse(dropped_random_forest_classifier, xs_imp, y), m_rmse(dropped_random_fores
t_classifier, valid_xs_imp, valid_y)
```

Out[38]:

```
(0.180822, 0.245178)
```

In [39]:

```
#Accuracy seems to have gotten worse ...
```

In [40]:

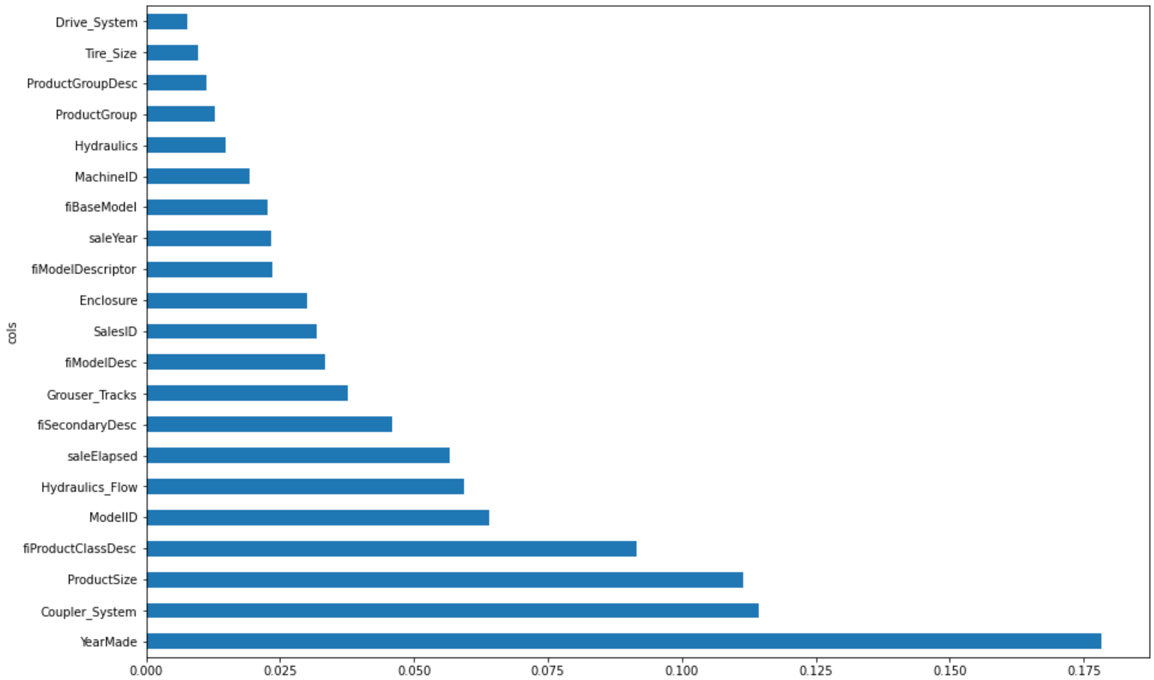
```
len(xs.columns), len(xs_imp.columns)
```

Out[40]:

```
(66, 21)
```

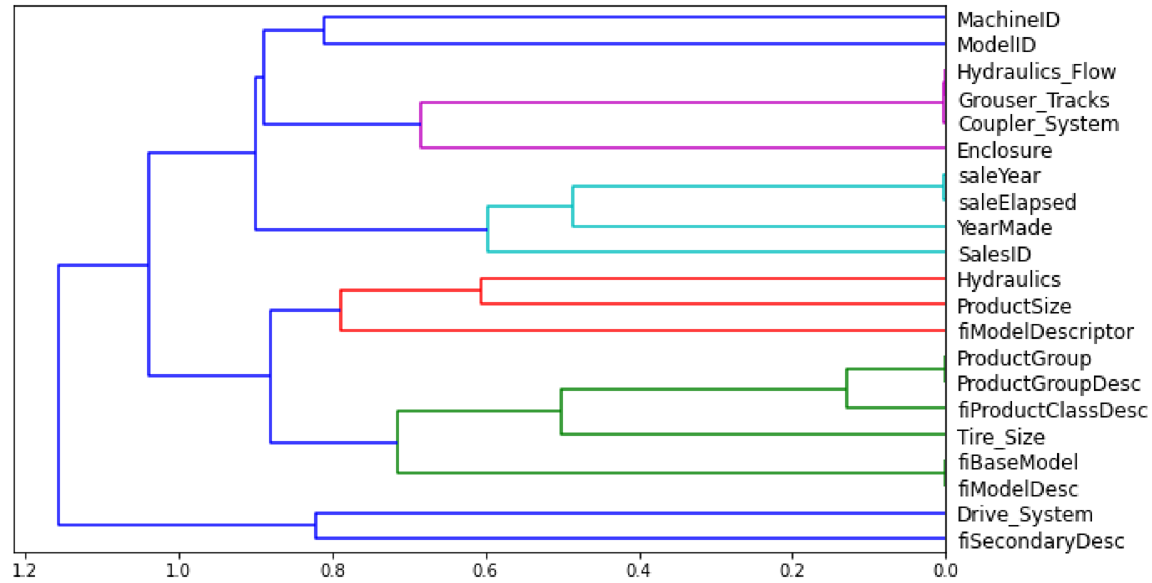
In [41]:

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



In [42]:

```
#Some categories have similar meanings and hence are just adding to the cloud ar
ound our data ...
#ProductGroup and ProductGroupDesc more or less tell us the same thing right ...
#Let's Find out

cluster_columns(xs_imp)
```



In [43]:

```
#As Shown in the plot above, we can see that some columns basically refer to the
same thing such as ProductGroup and ProductGroupDesc
#Logically we can remove those Columns too
```

In [44]:

```
#OOB score is a number returned by sklearn that ranges between 1.0 for a perfect
model and 0.0 for a random model
#Now lets create a function to create a Random Forest and and return us the OOB
 Score we are using a function as we
#don't necessarily want to create and keep the Random Forest but rather see how
 the removal of certain groups (groups we deem to
# be referring to the same thing ) affect our model.

def get_oob(xs_imp ,y):
    RandomForestInFunction = RandomForestRegressor(n_estimators=40, min_samples_
leaf=15,
        max_samples=50000, max_features=0.5, n_jobs=-1, oob_score=True)
    RandomForestInFunction.fit(xs_imp, y)

    oob_score  = RandomForestInFunction.oob_score_
    # print("oob_score_ : {oob_score}".format(oob_score=RandomForestInFunction.o
ob_score_) )
    return RandomForestInFunction.oob_score_
```

In [45]:

```
get_oob(xs_imp,y)
```

Out[45]:

```
0.8760086046887521
```

In [46]:

```
{"Removal of Column {c}".format(c=c):get_oob(xs_imp.drop(c, axis=1) , y) for c i
n (
    'saleYear', 'saleElapsed', 'ProductGroupDesc','ProductGroup',
    'fiModelDesc', 'fiBaseModel',
    'Hydraulics_Flow','Grouser_Tracks', 'Coupler_System')}
```

Out[46]:

```
{'Removal of Column Coupler_System': 0.8762282113523997,
 'Removal of Column Grouser_Tracks': 0.8767337561310908,
 'Removal of Column Hydraulics_Flow': 0.8766170664160736,
 'Removal of Column ProductGroup': 0.8762575473404198,
 'Removal of Column ProductGroupDesc': 0.8762674514269541,
 'Removal of Column fiBaseModel': 0.8752639699757805,
 'Removal of Column fiModelDesc': 0.8755873806077373,
 'Removal of Column saleElapsed': 0.8704392250751118,
 'Removal of Column saleYear': 0.8749191190775218}
```

In [47]:

```
#Let's try dropping multiple columns , specifically one of each of the columns w
e deemed to be very closely related to each other,
#Based on the cluster columns
to_drop = ['saleYear', 'ProductGroupDesc', 'fiBaseModel', 'Grouser_Tracks']
get_oob(xs_imp.drop(to_drop, axis=1) , y)
```

Out[47]:

```
0.8732722142605741
```

In [48]:

```
#Now , let's try dropping multiple columns , the ones that gave us the best oob_
score_ when removed from the dataframe

to_drop = ['Coupler_System', 'Grouser_Tracks', 'Hydraulics_Flow', 'ProductGroup'
]
get_oob(xs_imp.drop(to_drop, axis=1),y)
```

Out[48]:

```
0.8757139408888635
```

In [49]:

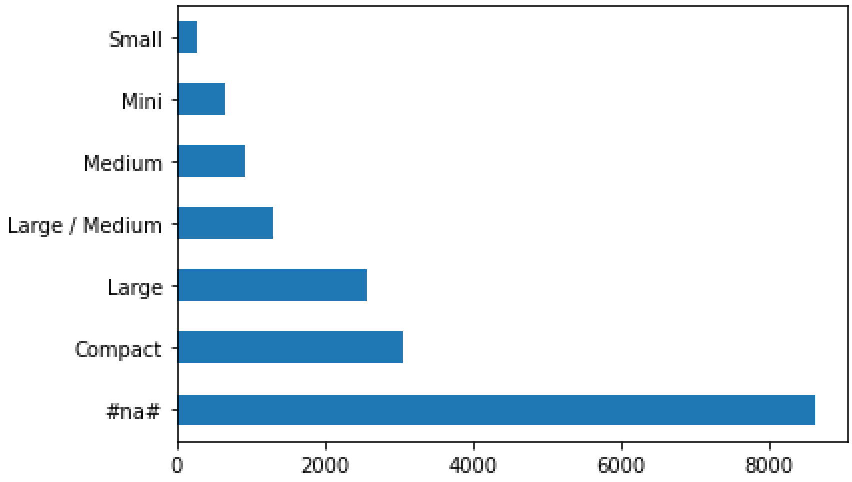
```
#since the columns that were dropped above gave us a better oob_score_ we drop t
hose and test against our validation set
to_drop = ['Coupler_System', 'Grouser_Tracks', 'Hydraulics_Flow', 'ProductGroup'
,'saleYear']
xs_final = xs_imp.drop(to_drop, axis=1)
valid_xs_final = valid_xs_imp.drop(to_drop, axis=1)
random_forest_classifier_with_columns_dropped = rf(xs_final, y)
m_rmse(random_forest_classifier_with_columns_dropped, xs_final, y), m_rmse(rando
m_forest_classifier_with_columns_dropped, valid_xs_final, valid_y)
```

Out[49]:

```
(0.181409, 0.245329)
```

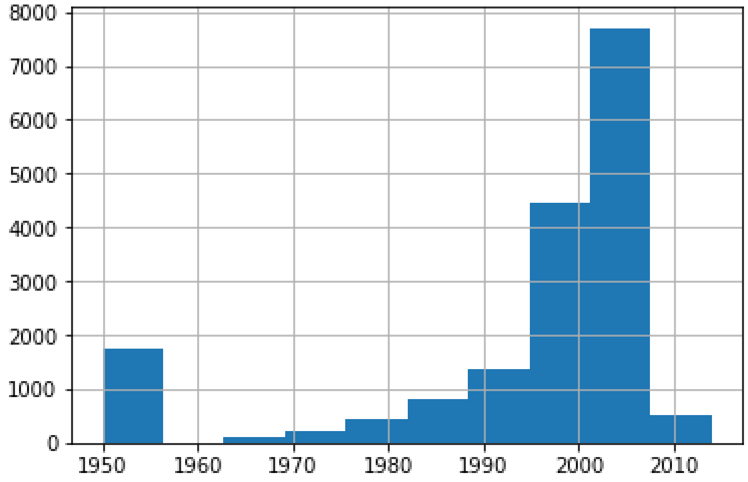
In [50]:

```
#We could also find if there is a dependency between certain categories and sale
price of the bulldozer


p = valid_xs_final['ProductSize'].value_counts(sort=True).plot.barh()
c = tt.classes['ProductSize']
plt.yticks(range(7), c);
#na is the label fastAi gives if we didnt specify the value for size
```



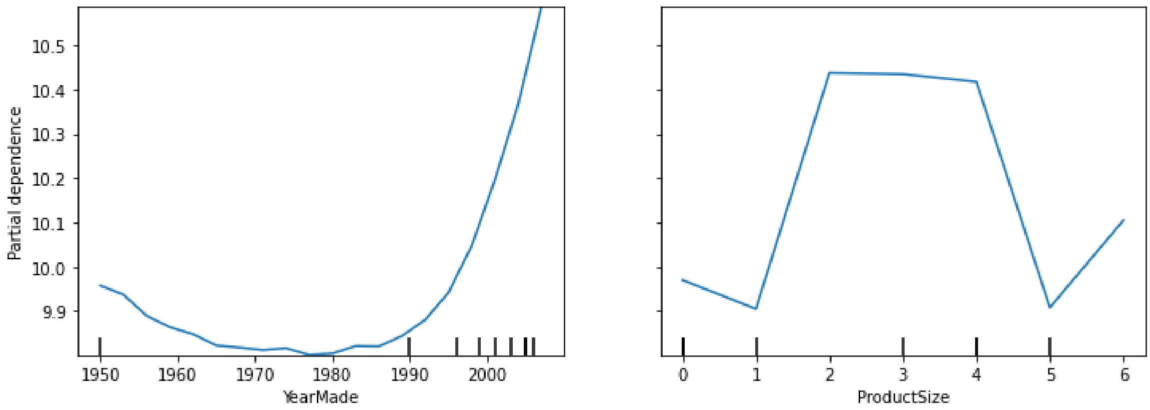
In [51]:

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



In [52]:

```
from sklearn.inspection import plot_partial_dependence

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



In [53]:

```
#What's going on Above ?
#Aren't we just taking the average price of all the predicted sales made in the
 given year and plotting it ?
#Well ... no, that tells us how all the fields(columns) affect the predicted pri
ce. However what we want to find is the
#effect of year on the selling price of the bulldozer at auction.

#To do this , we replace the value for year with the earliest year we have in th
e  "YearMade" column (we do this to all the vlaues)
# then we use our Random Forest predictor to predict the price of the bulldozer
 auction. With this we can see how the change in YearMade
#affects the predicted auction price ... hence the partial dependence of the Auc
tion price and the year that the bulldozer was made
```

In [54]:

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

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

In [55]:

```
#We have already deciphered how feature importance plays a part when we look at
 predicting across the whole dataset but we can do this
#for certain rows too (horiznotals i.e for a handfull of different tractors with
different attributes )

#We use a handfull of columns and start at the top of our first treee and trickl
e down, at each split we see which column was used to split the data
#we compare the model's prediction before and after the split and relate the Inc
rease or Decrease in the price to the column that
#was used as the binary splitter. We do this to all our 40 trees and add  the Im
provement/Disimprovement for all the columns in all the trees , normalize them
#so they sum up to one and we get the treeinterpreter as shown below.
```

In [56]:

```
row = valid_xs_final.iloc[:30]
```

In [57]:

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

In [58]:

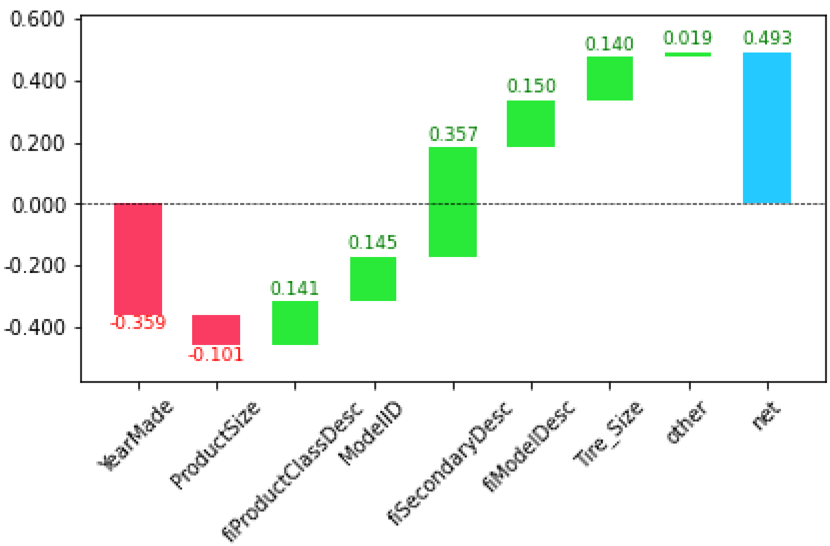
```
prediction[0], bias[0], contributions[0].sum() #for the first row
#Bias is the first prediction made by the 1st node of the Random Forest. Contrib
utuons are the changes in each predicition as we
#head down the Random Forest Tree the sum of the Contributions added to the Bias
will give us Predicition.
#Where Prediciton is the price of the auctioned off Bulldozer.
```

Out[58]:

```
(array([10.59414401]), 10.101273321533798, 0.492870690146211)
```

In [59]:

```
waterfall(valid_xs_final.columns, contributions[0], threshold=0.2, #threshold of
          rotation_value=45,formatting='{:,.3f}' );
```



In [60]:

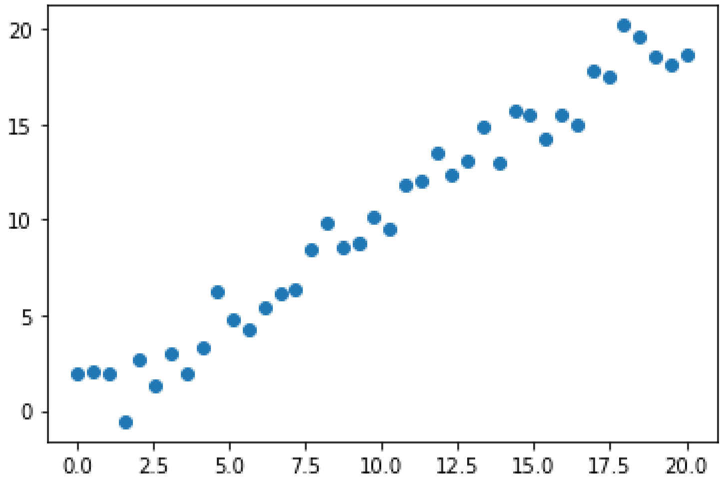
```
#We can use this after production to show users why a certain bulldozer was pred
icted to be auctioned off at a certain price
#given them insights as to why the Random Forest predicted the price as it di.
```

In [61]:

```
#The issue with Random Forest Regressors as with most ML algorithms is the fact
 that they dont perform too well with new data
```

In [62]:

```
np.random.seed(42)

x_lin = torch.linspace(0,20, steps=40)
y_lin = x_lin + torch.randn_like(x_lin)
plt.scatter(x_lin, y_lin);
```



In [63]:

```
xs_lin = x_lin.unsqueeze(1) #we need to give our random forect a matrix independ
ent variables, not a single vector
x_lin.shape,xs_lin.shape
```

Out[63]:

```
(torch.Size([40]), torch.Size([40, 1]))
```

In [64]:

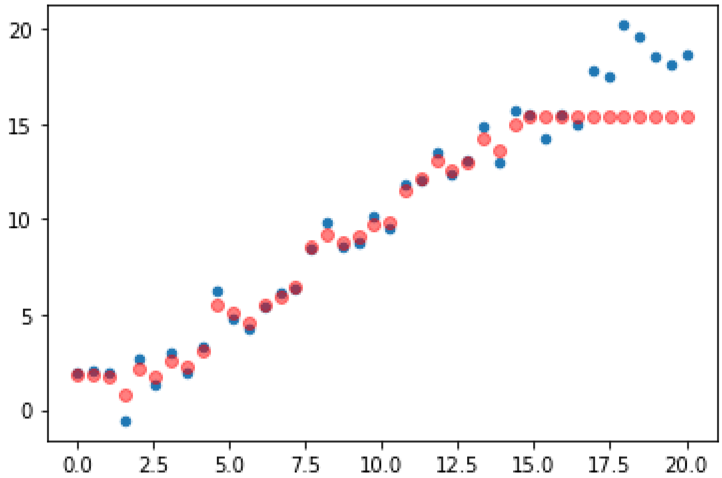
```
m_lin = RandomForestRegressor().fit(xs_lin[:30],y_lin[:30]) #use the first 30 da
ta points as our training set
```

In [65]:

```
plt.scatter(x_lin, y_lin, 20)
plt.scatter(x_lin, m_lin.predict(xs_lin), color='red', alpha=0.5)
```

Out[65]:

```
<matplotlib.collections.PathCollection at 0x7fa2bffa7a58>
```



In [66]:

```
#As Shown above , our random forest classifier does amazing in prediciting data
 it has come across or data similar to what it has
# seen before but when it sees brand new data , it is completely off
```

In [67]:

```
#As such when using Random Forest Regressors we need to make sure that our train
ing and validation sets are not worlds apart,
#... How do we do that ?
```

In [68]:

```
#Since Random Forest can do classification too , we could just merge both our tr
aining and validation sets , then
#create an array that acts as a binary label if a particular row is part of the
 training or validation set.
#We could then use our Random Forest to predict , based on the details in a row
 whether it is part of the training or validation set.
#Then we use feature importance to understand what columns the model uses to ide
ntify if the
#row is part of the training or validation set

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]
```

Out[68]:

	cols	imp
4	saleElapsed	0.829480
8	SalesID	0.142496
9	MachineID	0.025396
0	YearMade	0.001385
6	Enclosure	0.000410
3	ModelID	0.000394

In [69]:

```
#Before moving any further, recall that we used  the condition below to split th
e training and test set. Let's proceed with this in mind
# ***** cond = ((df.saleYear<=2010) | ((df.saleYear==2011) & (df.saleMonth<=1
0))) *****

#From what we see above , saleElapsed is used as the main identifier to see if t
he model is part of the training or validaiton set.
#saleElapsed tells us how long it has been since the sale took place , this is a
very good pointer as to whether the machine was
#sold before or after October 2011. salesID also seems to have some form of time
element attached
#to it. As for MachineID we could say that some models of bulldozers(identifier
 being their MachineID ) were only made after a certain
#year, since we used the saleYear to split the test and validation tests , a tra
ctor can only be sold during or after
#the year of its production .For example if you gave an Apple employee your phon
e's Serial Number
#he/she will be able to tell you what model of Iphone you're carrying and hence
 the year that it was
# released and thus he will be at least able to predict with decent accuracy, th
e years you bought the phone (you can only buy it
#on the year of release or the years after)..
```

In [70]:

```
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 0.24589
SalesID 0.244059
saleElapsed 0.248706
MachineID 0.244209
```

In [71]:

```
time_vars = ['SalesID','MachineID']
#We remove SalesID and MachineID since it gives us better accuracies
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)
```

Out[71]:

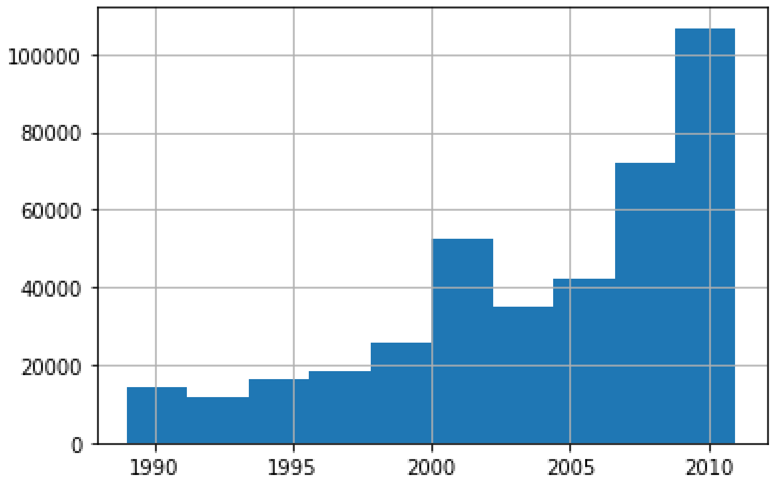
```
0.242036
```

In [72]:

```
#Another thing that can help is avoiding old data , old data ususally provides r
elationships that may not be true anymore...
```

In [73]:

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



In [74]:

```
filt_year = xs['saleYear']>2004#let's only keep data after 2004
xs_filt_year= xs_final_time[filt_year]
y_filt_year = y[filt_year]
```

In [75]:

```
m = rf(xs_filt_year, y_filt_year)
m_rmse(m, xs_filt_year, y_filt_year), m_rmse(m, valid_xs_time, valid_y)
```

Out[75]:

```
(0.175036, 0.243367)
```

In [75]:

```
```