Published in Towards Data Science

Col Jung Follow

Nov 14, 2020 · 14 min read ★ · Listen

Predict House Prices with Machine Learning

Regression model trained on 1,883 homes

Image source: Greg Carter Barrister & Solicitor.

Property valuation is an imprecise science. Individual appraisers and valuers bring

Perhaps a well-trained n e of a human, with greater consistency and accuracy.

Let's prototype this idea and train some ML models using data about a house's ***features***, ***costs*** and ***neighbourhood profile*** to predict its value. Our target variable — **property price** — is numerical, hence the ML task is **regression**. (For a categorical target, the task becomes classification.)

We'll use a dataset from elitedatascience.com that simulates a portfolio of 1,883 properties belonging to a real-estat t (REIT). There are 26 columns. Here's a small snippet:

	tx_price	beds	baths	sqft	year_built	lot_size	property_type	exterior_walls	roof	basement	restaurants	groceries	nightlife
0	295850	1	1	584	2013	0	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	107	9	30
1	216500	1	1	612	1965	0	Apartment / Condo / Townhouse	Brick	Composition Shingle	1.0	105	15	6
2	279900	1	1	615	1963	0	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	183	13	31
3	379900	1	1	618	2000	33541	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	198	9	38
4	340000	1	1	634	1992	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	149	7	22
5	265000	1	1	641	1947	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	146	10	23
6	240000	1	1	642	1944	0	Single-Family	Brick	NaN	NaN	159	13	36
7	388100	1	1	650	2000	33541	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	198	9	38
8	240000	1	1	660	1983	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	51	8	6
9	250000	1	1	664	1965	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	119	10	26

cafes	shopping	arts_entertainment	beauty_spas	active_life	median_age	married	college_grad	property_tax	insurance	median_school	num_schools	tx_year
19	89	6	47	58	33.0	65.0	84.0	234.0	81.0	9.0	3.0	2013
13	87	2	26	14	39.0	73.0	69.0	169.0	51.0	3.0	3.0	2006
30	101	10	74	62	28.0	15.0	86.0	216.0	74.0	8.0	3.0	2012
25	127	11	72	83	36.0	25.0	91.0	265.0	92.0	9.0	3.0	2005
20	83	10	50	73	37.0	20.0	75.0	88.0	30.0	9.0	3.0	2002
27	86	9	60	52	28.0	15.0	86.0	168.0	58.0	8.0	3.0	2004
17	92	12	66	50	28.0	36.0	88.0	176.0	61.0	7.0	3.0	2011
25	127	11	72	83	36.0	25.0	91.0	266.0	92.0	9.0	3.0	2005
2	40	18	32	41	36.0	49.0	77.0	188.0	65.0	6.0	3.0	2013
25	183	13	70	36	57.0	13.0	83.0	147.0	51.0	6.0	3.0	2007

Snapshot of the original dataset.

The steps are:

1. **EDA & data-processing:** explore, visualise and clean the data.
2. **Feature engineering:** leverage domain expertise and create new features.

4. **Performance evalu** task metrics like the R^2-score and mean squared average.

5. **Deployment:** batch-run or get some data engineers / ML engineers to build an automated pipeline?

1. Data exploration and processing

Exploratory data analysis (EDA) helps us understand the data and provides ideas and insights for **data cleaning** and **feature engineering**. Data cleaning prepares the data for our algorithms while feature engineering is the magic sauce that will really help our algorithms draw out the underlying patterns from the dataset. Remember:

> Better data always beats fancier algorithms!

We start by loading some standard data science Python packages into JupyterLab.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb

from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso, Ridge, ElasticNet
from sklearn.ensemble import RandomForestRegressor,
                             GradientBoostingRegressor

from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV

from sklearn.metrics import r2_score,
                            mean_absolute_error

import pickle
```

Here's a snapshot of our dataframe again. The shape is (1,883, 26).

	tx_price	beds	baths	sqft	year_built	lot_size	property_type	exterior_walls	roof	basement	restaurants	groceries	nightlife	cafes	shopping	arts_entertainment	beauty_spas	active_life	median_age	married	college_grad	property_tax	insurance	median_school	num_schools	tx_year
0	295850	1	1	584	2013	0	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	107	9	30	19	89	6	47	58	33.0	65.0	84.0	234.0	81.0	9.0	3.0	2013
1	216500	1	1	612	1965	0	Apartment / Condo / Townhouse	Brick	Composition Shingle	1.0	105	15	6	13	87	2	26	14	39.0	73.0	69.0	169.0	51.0	3.0	3.0	2006
2	279900	1	1	615	1963	0	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	183	13	31	30	101	10	74	62	28.0	15.0	86.0	216.0	74.0	8.0	3.0	2012
3	379900	1	1	618	2000	33541	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	198	9	38	25	127	11	72	83	36.0	25.0	91.0	265.0	92.0	9.0	3.0	2005
4	340000	1	1	634	1992	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	149	7	22	20	83	10	50	73	37.0	20.0	75.0	88.0	30.0	9.0	3.0	2002
5	265000	1	1	641	1947	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	146	10	23	27	86	9	60	52	28.0	15.0	86.0	168.0	58.0	8.0	3.0	2004
6	240000	1	1	642	1944	0	Single-Family	Brick	NaN	NaN	159	13	36	17	92	12	66	50	28.0	36.0	88.0	176.0	61.0	7.0	3.0	2011
7	388100	1	1	650	2000	33541	Apartment / Condo / Townhouse	Wood Siding	NaN	NaN	198	9	38	25	127	11	72	83	36.0	25.0	91.0	266.0	92.0	9.0	3.0	2005
8	240000	1	1	660	1983	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	51	8	6	2	40	18	32	41	36.0	49.0	77.0	188.0	65.0	6.0	3.0	2013
9	250000	1	1	664	1965	0	Apartment / Condo / Townhouse	Brick	NaN	NaN	119	10	26	25	183	13	70	36	57.0	13.0	83.0	147.0	51.0	6.0	3.0	2007

Snapshot of the original dataset. Click to expand.

The target variable is **tx_price**, which represents the price at which the property was last sold.

There are 25 columns/features.

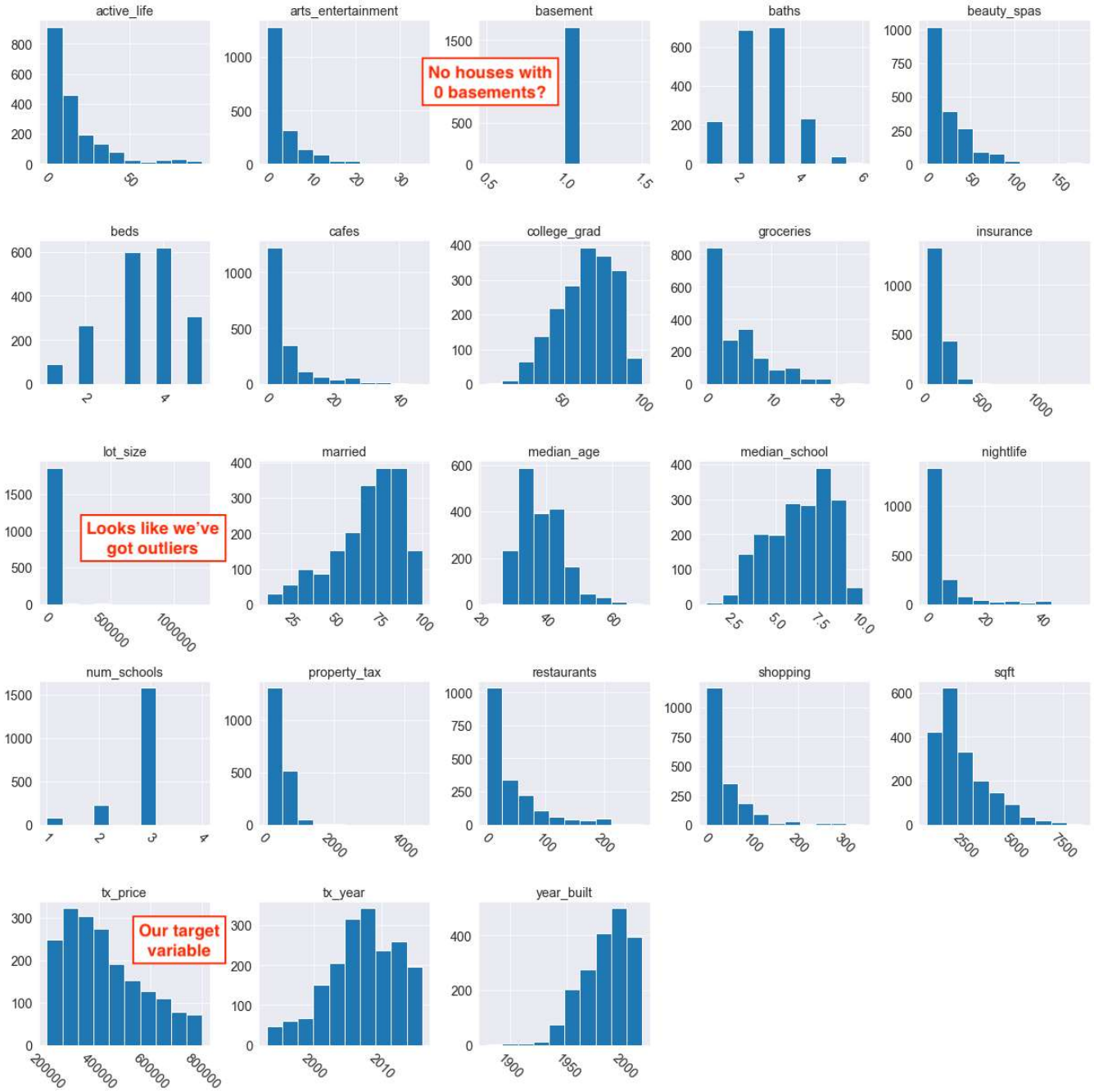
- Property characteristics: **tx_year** — year when it was last sold, **property_type** (single-family vs. apartment/condo/townhouse).
- Property costs: monthly **property_tax** and **insurance**.
- Property features: **beds, baths**, **sqrt** (floor area), **lot_size** (incl. outside area), **year_built** & **basement** (yes/no).
- Neighbourhood lifestyle: number of **restaurants**, **groceries**, **nightlife**, **cafes**, **shopping**, **arts_entertainment**, **beauty spas** & **active_life** (gym, yoga studios etc.) within 1 mile radius.
- Neighbourhood demographic: **median_age**, **married** (%), **college_grad** (%).
- Neighbourhood schools: **num_schools** (within district) & **median_schools** (median score out of 10 of public schools in district).

1.1 Numerical features

A list of our numerical features can be obtained with the code:

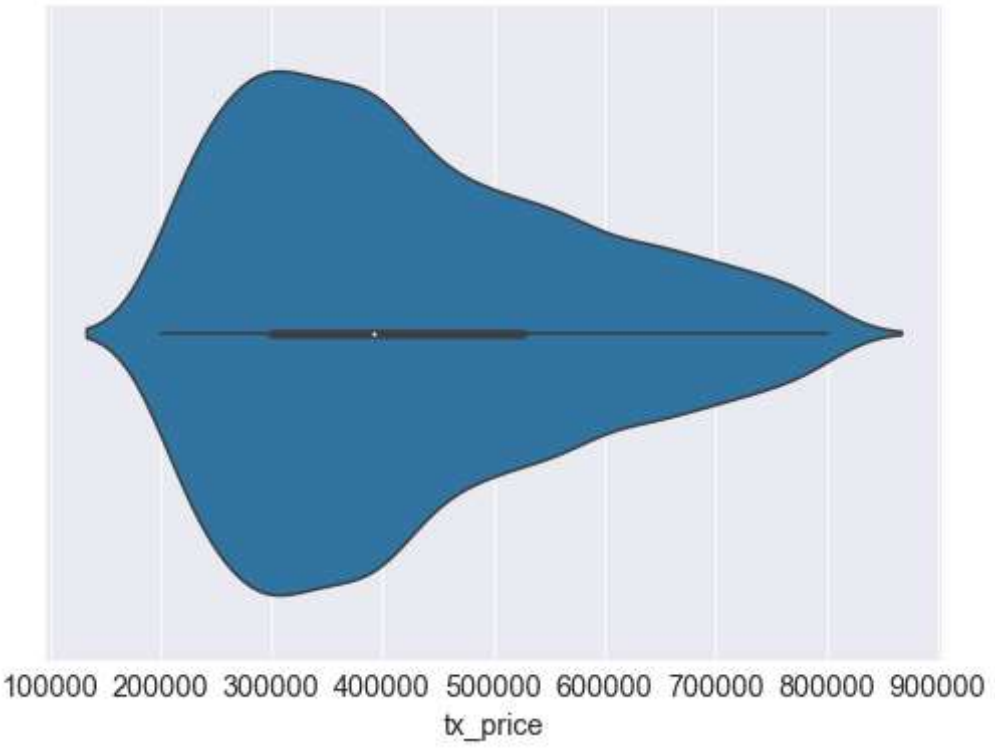
We'll skip that here. Inst ams for all our numerical features:

```
df.hist(figsize=(20,20), xrot=-45)
```



This looks mostly fine. I've left a few comments in red.

```
sns.violinplot(dat
```




```
df.tx_price.median()
```

Output: 392000

The median US house price in 2020 Oct is $325,000, so the houses amassed in this REIT portfolio appears to be a bit more uptown on average.

Missing values:

```
df.select_dtypes(exclude=['object']).isnull().sum()
```

Output:

tx_price	**0**
beds	**0**
baths	**0**
sqft	**0**
year built	**0**

```
cafes                 0
shopping              0
arts_entertainment    0
beauty_spas           0
active_life           0
median_age            0
married               0
college_grad          0
property_tax          0
insurance             0
median_school         0
num_schools           0
tx_year               0
```

A closer examination of the **basement** feature suggests that rows with basement=0 were encoded with NaN. Hence this is actually an **incorrect labelling** problem. We'll need to convert NaN's to 0 for the algorithms.

```
df.basement.fillna(0, inplace=True)
```

Note: If these NaN's were genuine missing values, we should create an indicator variable ***basement_missing*** *(with value 1 when basement=NaN) before converting the NaNs in* ***basement*** *to 0.*

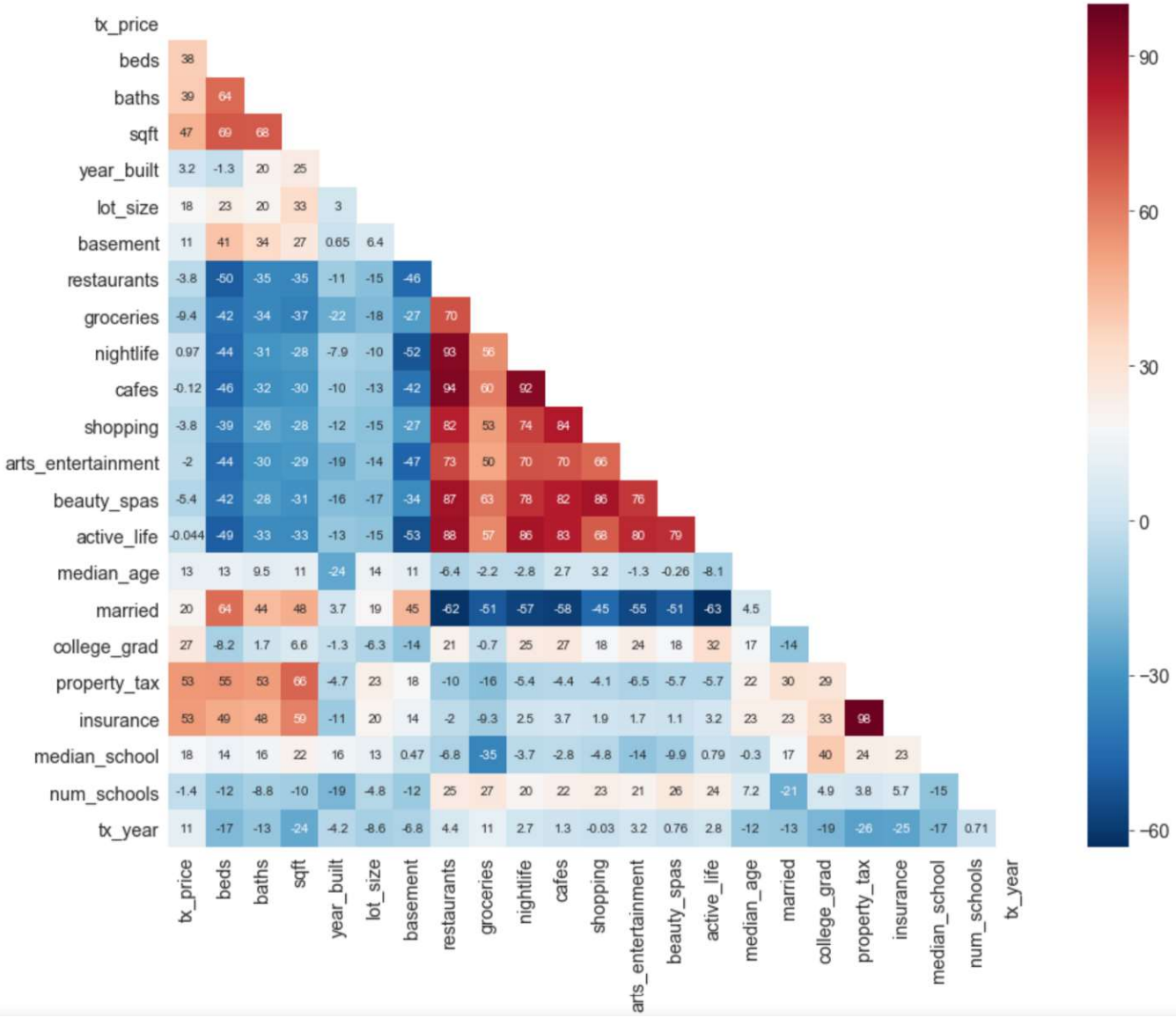
Outliers: The histogram for **lot_size** suggests we have some pretty large mansion estates!

```
df.lot_size.sort_values(ascending=False)

Output:
102     1220551
1111     436471
1876     436035
1832     436035
1115     435600
Name: lot_size, dtype: int64
```

Actually, there's just one property with a lot size way bigger than the others. Let's

Here's a **correlation heatmap** for our numerical features.

```
# mask out upper triangle
mask = np.zeros_like(df.corr(), dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# heatmap
sb.heatmap(df.corr()*100,
           cmap='RdBu_r',
           annot = True,
           mask = mask)
```



nightlife and **restauran** ollinearity might affect model performance, as regression features should be independent.

1.2 Categorical features

Our categorical variables can be listed with the code:

```
df.dtypes[df.dtypes=='object']
```

These are: **property_type**, **exterior_walls** and **roof.**

The classes for each of these categorical features can be listed with:

```
df.property_type.value_counts()

Output:
Single-Family                   1080
Apartment / Condo / Townhouse    802
Name: property_type, dtype: int64

df.exterior_walls.value_counts()

Output:
Brick                  686
Siding (Alum/Vinyl)    503
Metal                  120
Combination            107
Wood                    72
Wood Siding             49
Brick veneer            48
Stucco                  26
Other                   10
Concrete                 8
Concrete Block           7
Block                    7
Asbestos shingle         6
Rock, Stone              5
Masonry                  3
Wood Shingle             2
```

```
Output:
Composition Shingle      1179
Asphalt                   132
Shake Shingle              55
Other                      49
Wood Shake/ Shingles       30
Gravel/Rock                30
Roll Composition           12
Asbestos                    9
Slate                       9
Composition                 5
asphalt                     5
Metal                       4
composition                 4
shake-shingle               3
Built-up                    2
asphalt,shake-shingle       1
Name: roof, dtype: int64
```

From this, we can clean up the classes a bit. We'll

- merge together sparse classes (those with too few observations)
- merge classes with similar meanings (e.g. subsume *Concrete* and *Block* into the more general *Concrete block* class.
- fix up labelling errors (e.g. *concrete* should be *Concrete*).

```
df.exterior_walls.replace(['Wood Siding', 'Wood Shingle', 'Wood'],
                          'Wood', inplace=True)

df.exterior_walls.replace('Rock, Stone', 'Masonry', inplace=True)

df.exterior_walls.replace(['Concrete','Block'], 'Concrete Block',
                           inplace=True)

df.exterior_walls.replace(['Concrete Block', 'Stucco', 'Concrete',
                          'Block', 'Masonry', 'Other',
                          'Asbestos shingle', 'Rock, Stone'],
                          'Other', inplace=True)

df.roof.replace(['Composition', 'Wood Shake/ Shingles',
                 'Composition Shingle'], 'Composition Shingle',
                  inplace=True)
```


```
df.roof.replace(['                                    ,
                 'Shake Shingle'], 'Shake Shingle', inplace=True)

df.roof.replace('composition', 'Composition',inplace=True)
```

Missing values:

```
df.select_dtypes(include=['object']).isnull().sum()
```

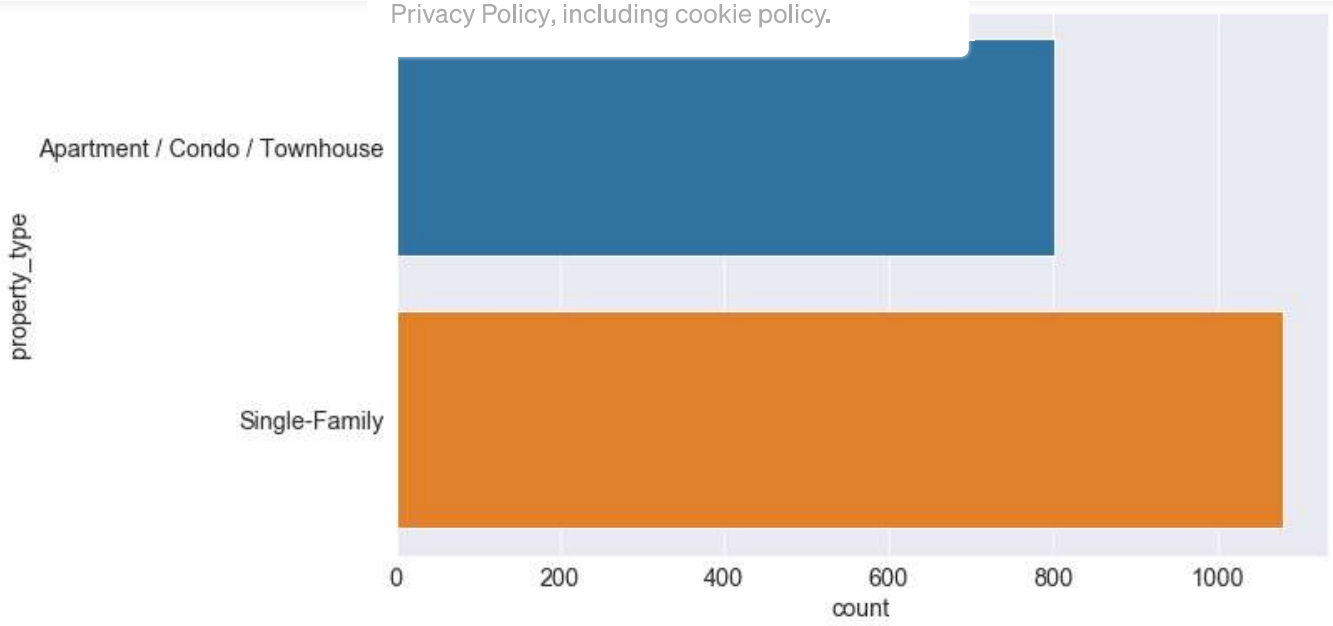
```
Output:
property_type       0
exterior_walls    223
roof              353
dtype: int64
```

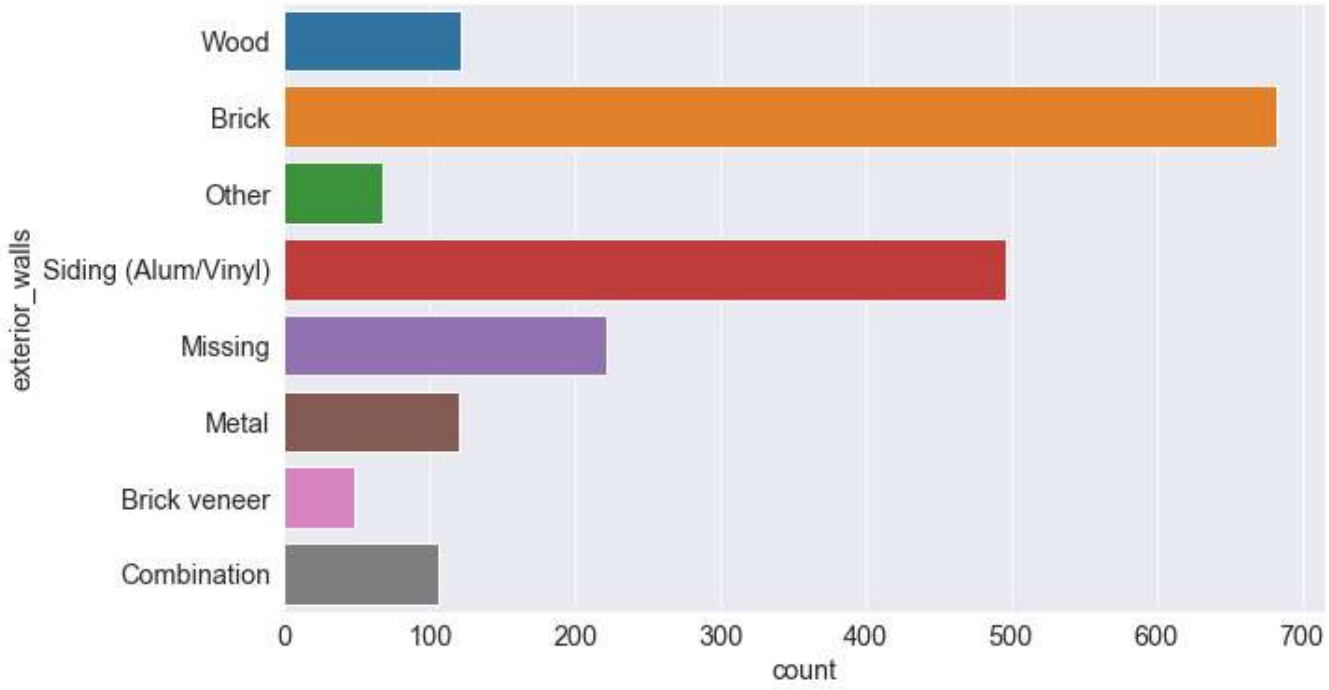
We'll want to tell our algorithms that these values are *Missing*. This is more instructive than simply removing the rows.

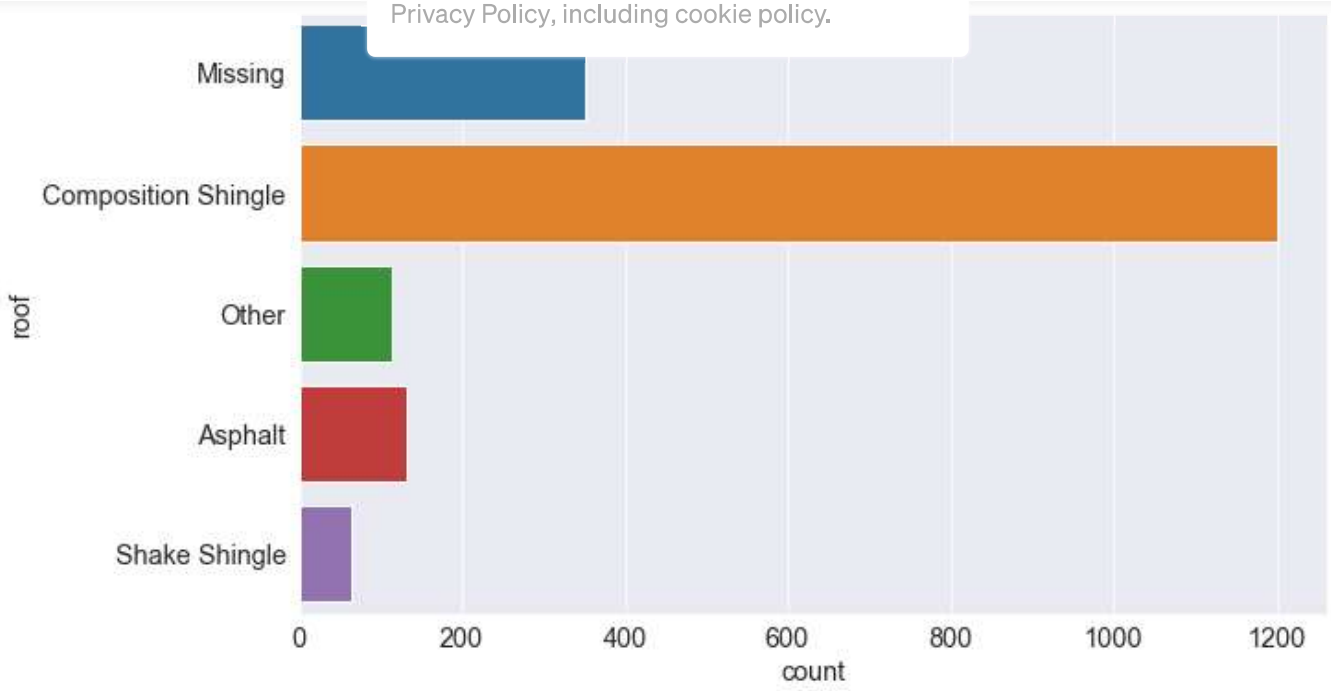
```
for feat in df.select_dtypes(include=['object']):
    df[feat] = df[feat].fillna("Missing")
```

Let's plot some bar graphs of our three categorical features.

```
for feat in df.dtypes[df.dtypes=='object'].index:
   sb.countplot(y=feat, data=df)
```







Finally, categorical features must be one-hot encoded for our algorithms. We'll do that now.

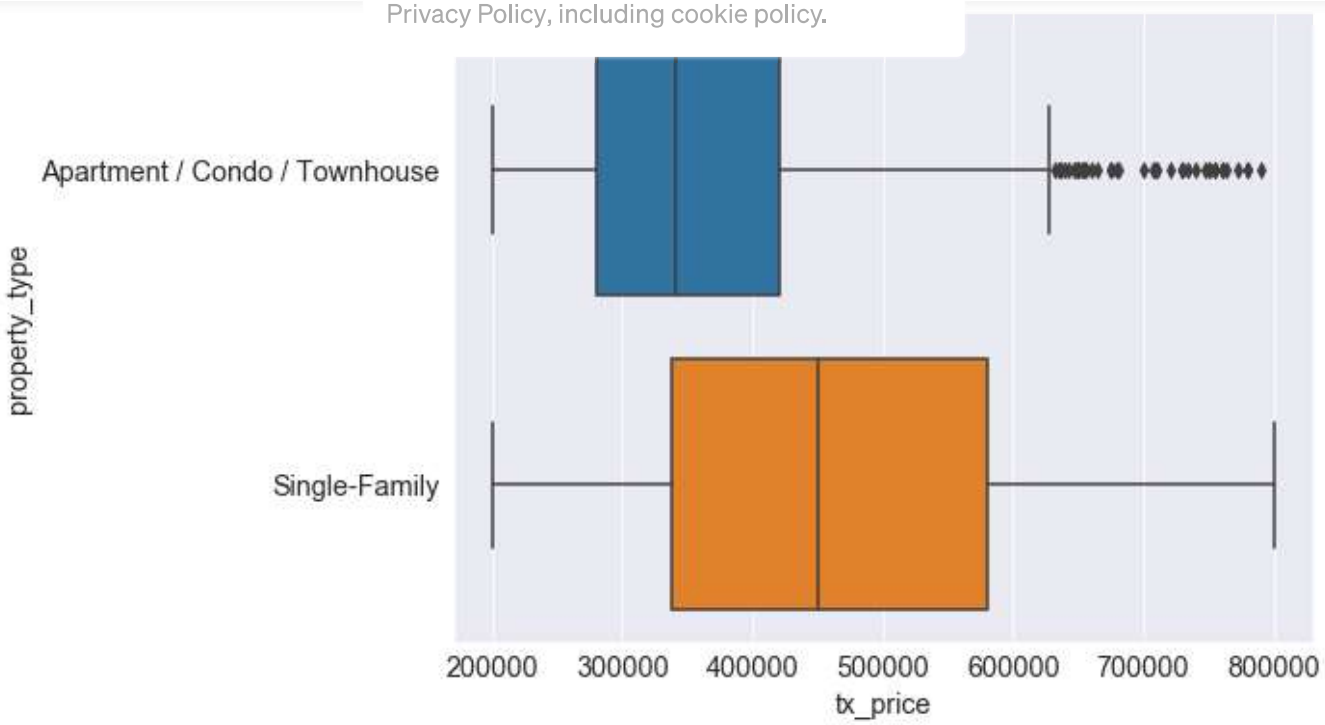
```
df = pd.get_dummies(df, columns = ['exterior_walls',
                                   'roof',
                                   'property_type'])
```

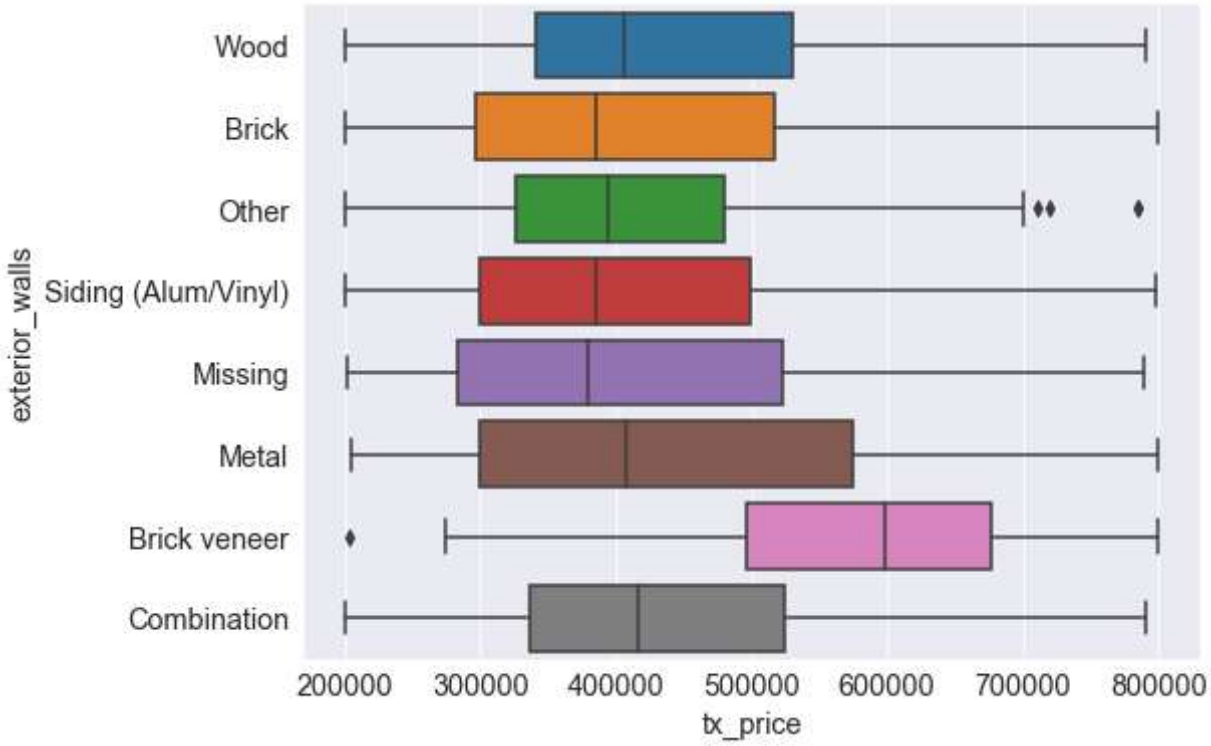
1.3 Segmentations

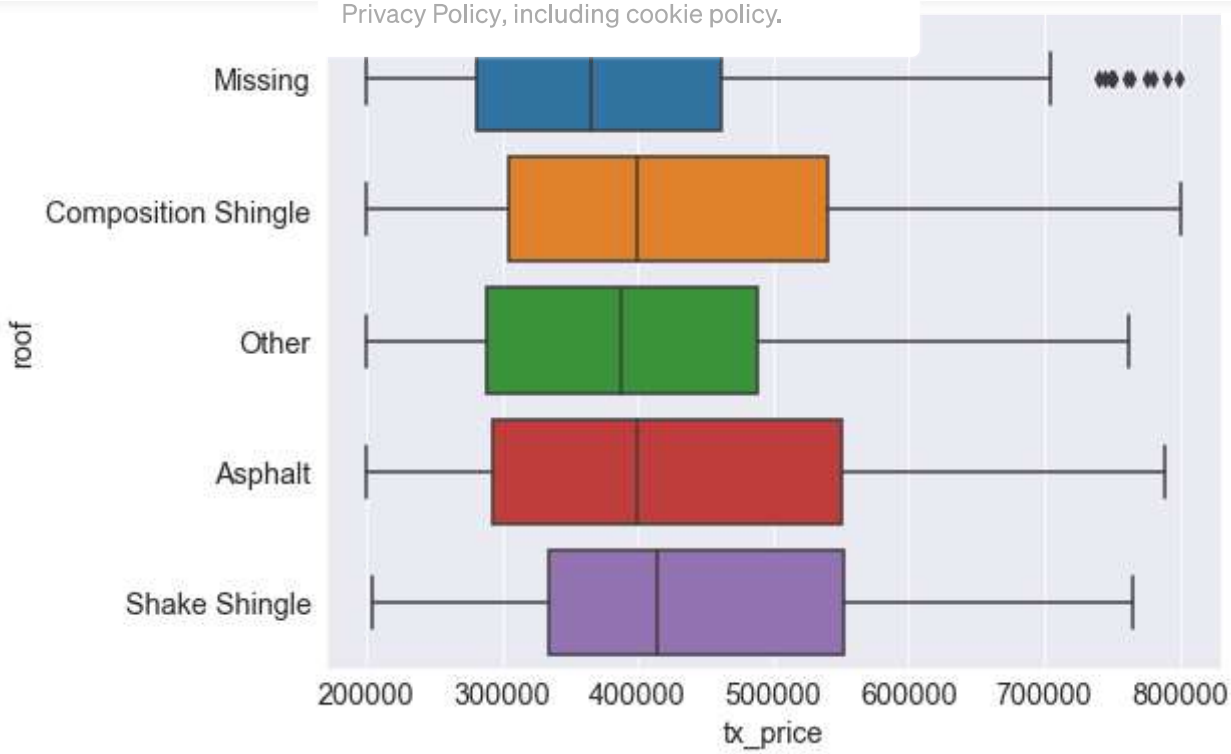
Segmentations combine both numerical and categorical features.

Let's segment all our categorical variables (**property_type**, **exterior_walls** and **roof**) by our target **tx_price**. This will provide a detailed look at what might drive property values.

```
for feat in df.dtypes[df.dtypes=='object'].index:
   sb.boxplot(data=df, x = 'tx_price', y = '{}'.format(feat))
```







Let's also look at how property and neighbourhood features differ between single-family homes and apartment/condo/townhouse by using a **groupby**.

```
df.groupby('property_type').agg(['mean','median'])
```

	tx_price		beds		baths		sqft		year_built		lot_size		basement		restaurants		groceries		nightlife		cafes	
	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median
property_type																						
Apartment / Condo / Townhouse	366705.824190	340000	2.602244	3	2.201995	2	1514.523691	1428	1988.985037	1991	2427.273067	1424	0.784289	1.0	58.412718	42	5.912718	5	7.851621	4	8.037406	5
Single-Family	465570.491989	450000	4.018850	4	2.837889	3	2892.283695	2640	1977.983035	1980	20152.588124	10724	0.950990	1.0	26.701225	14	3.459943	2	3.021678	1	3.316682	1

shopping		arts_entertainment		beauty_spas		active_life		median_age		married		college_grad		property_tax		insurance		median_school		num_schools	
mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median
57.631382	36	4.840598	3	32.087173	26	22.410959	14	37.199253	35.0	57.534247	62.0	66.372354	69.0	346.261519	321.0	105.652553	96.0	6.382316	7.0	2.831880	3.0
28.289815	13	2.318519	1	16.970370	9	10.946296	7	39.643519	40.0	77.685185	80.0	64.128704	65.0	556.383333	516.0	166.329630	152.5	6.592593	7.0	2.764815	3.0

In summary, single-family houses are more expensive and larger, while apartment/condo/townhouse dwellings are closer to entertainment/shops and attract younger residents. This isn't too surprising. Family homes tend to be in suburbia, while apartments and higher density living tend to be closer to downtown.

A lot of feature engineer ave a subject matter expert (SME) on real estate to provide guidance, you'll have a better chance of engineering some awesome feature that will really make your modelling shine.

Here, we'll engineer four new features:

- **two_and_two**: Properties with two bedrooms and two bathrooms. Why? Through domain expertise, you might know that these properties are popular with investors. This is an indicator variable.
- **during_recession**: The US property market went into recession between 2010 and 2013. Thus being in this time period might influence pricing to a substantial degree.
- **property_age**: This one is kind of common sensical. Newer properties should attract a higher price, right? (Answer: yes but not always.)
- **school_score**: We define this *interaction feature* to be the product of **num_school** and **median_school**, which gives a single score that represents the quality of schooling in the area.

The following code creates these new features.

```
df['two_and_two'] = ((df.beds == 2) & (df.baths == 2)).astype(int)

df['during_recession'] = ((df.tx_year >= 2010) &
                          (df.tx_year <= 2013))
                          .astype(int)

df['property_age'] = df.tx_year - df.year_built
df.drop(['tx_year', 'year_built'], axis=1, inplace=True)

df['school_score'] = df.num_schools * df.median_school
```

The new **property_age** feature arguably supercedes the original **tx_year** and **year_built**, thus we'll remove them.

```
df.two_and_two.mean()
df.during_recession.mean()

Outputs:
0.09458023379383634
0.2635494155154091
```

Analytical base table: The dataset after applying all of these data cleaning steps and feature engineering is our **analytical base table**. This is the data on which we train our models.

Our ABT has 1,863 properties and 40 columns. *Recall our original dataset had just 26 columns!* Generally, ABTs have more columns than the original dataset because of:

- one-hot encoding, where a new column is created for *every* class in *every* categorical feature; and
- feature engineering.

On a related note, a problem in ML is the curse of dimensionality, where your ABT has too many columns/features. This is a major problem in deep learning, where data processing can result in ABTs with thousands of features or more. **Principal component analysis (PCA)** is a dimensionality-reduction technique where high dimensional correlated data is transformed to a lower dimensional set of uncorrelated components, referred to as **principal components** (PC). The good news is that the lower dimensional PCs capture most of the information in the high dimensional dataset.

I plan to write an article on unsupervised learning techniques, including PCA. Keep your eyes peeled!

3. Modelling

We're going to train four tried-and-true regression models:

- **gradient-boosted t**

First, let's split our analytical base table.

```
y = df.status
X = df.drop('tx_price', axis=1)
```

We'll then split into training and test sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=1234)
```

We'll set up a **pipeline object** to train. This will streamline our model training process.

```
pipelines = {
    'lasso' : make_pipeline(StandardScaler(),
              Lasso(random_state=123)),
    'ridge' : make_pipeline(StandardScaler(),
              Ridge(random_state=123)),
    'enet' :  make_pipeline(StandardScaler(),
              ElasticNet(random_state=123)),
    'rf' :    make_pipeline(
              RandomForestRegressor(random_state=123)),
    'gb' :    make_pipeline(
              GradientBoostingRegressor(random_state=123))
}
```

We also want to tune the **hyperparameters** for each algorithm.

For all three regularised regressions, we'll tune **alpha** (L1 & L2 penalty strengths), along with the **l1_ratio** for Elastic Net (i.e. weighting between L1 & L2 penalties).

```
lasso_hyperparameters = {
    'lasso__alpha' : [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10]}
```


```
    'elasticnet__l                                    0.9]}
```

For our random forest, we'll tune the number of estimators (**n_estimators**) and the max number of features to consider during a split (**max_features**), and the min number of samples to be a leaf (**min_samples_leaf**).

```
rf_hyperparameters = {
     'randomforestregressor__n_estimators' : [100, 200],
     'randomforestregressor__max_features' : ['auto', 'sqrt', 0.33],
     'randomforestregressor__min_samples_leaf' : [1, 3, 5, 10]}
```

For our gradient-boosted tree, we'll tune the number of estimators (**n_estimators**), **learning rate**, and the maximum depth of each tree (**max_depth**).

```
gb_hyperparameters = {
      'gradientboostingregressor__n_estimators' : [100, 200],
      'gradientboostingregressor__learning_rate' : [0.05, 0.1, 0.2],
      'gradientboostingregressor__max_depth' : [1, 3, 5]}
```

Finally, we'll fit and tune our models. Using **GridSearchCV** we can train all of these models with cross-validation on all of our declared hyperparameters with just a few lines of code!

```
fitted_models = {}
for name, pipeline in pipelines.items():
    model = GridSearchCV(pipeline,
                         hyperparameters[name],
                         cv=10,
                         n_jobs=-1)
    model.fit(X_train, y_train)
    fitted_models[name] = model
```


4.1 Performance scores

We'll start by printing the **cross-validation scores**. This is the average performance across the 10 hold-out folds and is a way to get a reliable estimate of the model performance **using only your training data**.

```
for name, model in fitted_models.items():
    print( name, model.best_score_ )

Output:
lasso 0.3085486180300333
ridge 0.3165464682513239
enet 0.34280536738492506
rf 0.4944720180590308
gb 0.48797200970900745
```

Moving onto the **test data**, we'll output the **R2-score** and **mean absolute error (MAE)**.

The R^2-score represents the proportion of total variance explained by the model and ranges from 0 to 100. If the R^2-score = 100, then the dependent variable (**tx_price**) correlates perfectly with the features.

The MAE is the average error between the predictions and actuals.

```
for name, model in fitted_models.items():
   pred = model.predict(X_test)
   print(name)
   print(' - - - - ')
   print('R²:', r2_score(y_test, pred))
   print('MAE:', mean_absolute_error(y_test, pred))
   print()

Output:
lasso
--------
R^2: 0.4088031693011063
MAE: 85041.97658598644

ridge
--------
```

```
R^2: 0.40522476546
MAE: 86297.6516160
```

```
rf
--------
R^2: 0.5685576834419455
MAE: 68825.53227240045
```

```
gb
--------
R^2: 0.5410951822821564
MAE: 70601.60664940192
```

The winning algorithm is the random forest, with the highest R^2 score of 0.57 and the lowest MAE. We can actually do better than this.

Remember earlier we removed the **tx_year** and **year_built** features after engineering **property_age**? It turns out that was the wrong choice. Including them would have given the model a big performance bump to $R^2 = 0.81$. Moreover, leaving out a few of the highly correlated neighbourhood profile features (i.e. **active_life, beauty_spas, cafe, nightlife** and **restaurants**) would have increased performance even more. This highlights the importance of both **feature engineering** and **feature selection.**

FYI, here are the hyperparameters of the winning random forest, tuned using GridSearchCV.

```
RandomForestRegressor(bootstrap=True,
                      criterion='mse',
                      max_depth=None,
                      max_features='auto',
                      max_leaf_nodes=None,
                      min_impurity_decrease=0.0,
                      min_impurity_split=None,
                      min_samples_leaf=10,
                      min_samples_split=2,
                      min_weight_fraction_leaf=0.0,
                      n_estimators=200,
                      n_jobs=None,
                      oob_score=False,
                      random_state=123,
                      verbose=0,
                      warm_start=False)
```

4.2 Feature importances

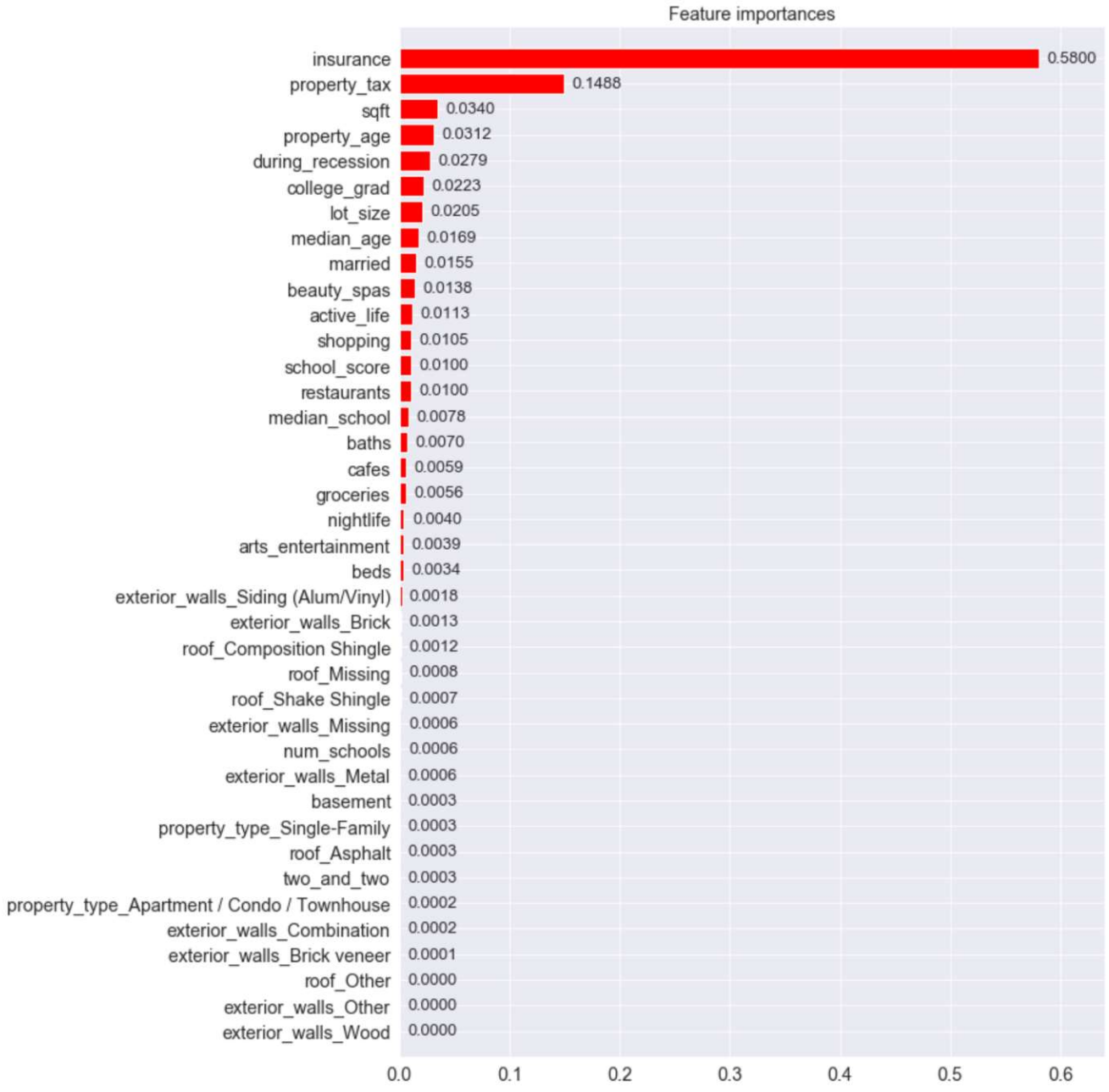
Consider the following code.

```
coef = winning_model.feature_importances_
ind = np.argsort(-coef)

for i in range(X_train.shape[1]):
    print("%d. %s (%f)" % (i + 1, X.columns[ind[i]], coef[ind[i]]))

x = range(X_train.shape[1])
y = coef[ind][:X_train.shape[1]]

plt.title("Feature importances")
ax = plt.subplot()
plt.barh(x, y, color='red')
ax.set_yticks(x)
ax.set_yticklabels(X.columns[ind])
plt.gca().invert_yaxis(
```

This will print a list of features ranked by importance and a corresponding bar plot.

```
1. insurance (0.580027)
2. property_tax (0.148774)
3. sqft (0.033958)
4. property_age (0.031218)
5. during_recession (0.027909)
6. college_grad (0.022310)
7. lot_size (0.020546)
8. median_age (0.016930)
9. married (0.015506)
10. beauty_spas (0.013840)
11. active_life (0.011257)
12. shopping (0.010523)
13. school_score (0.010032)
14. restaurants (0.009975)
15. median_school (0.007809)
16. baths (0.007009)
17. cafes (0.005914)
18. groceries (0.005578)
19. nightlife (0.004049)
20. arts_entertainment (0.003944)
21. beds (0.003364)
22. exterior_walls_Siding (Alum/Vinyl) (0.001808)
```

```
28. num_schools (0
29. exterior_walls_Metal (0.000378)
30. basement (0.000348)
```



The top two predictors by far are

- cost of monthly homeowner's **insurance** and
- monthly **property tax**.

The next two strongest p qft) and how old it is (**property_age**). Larger and newer properties tend to fetch more in the market, hence these results conform to expectations too.

5. Deployment

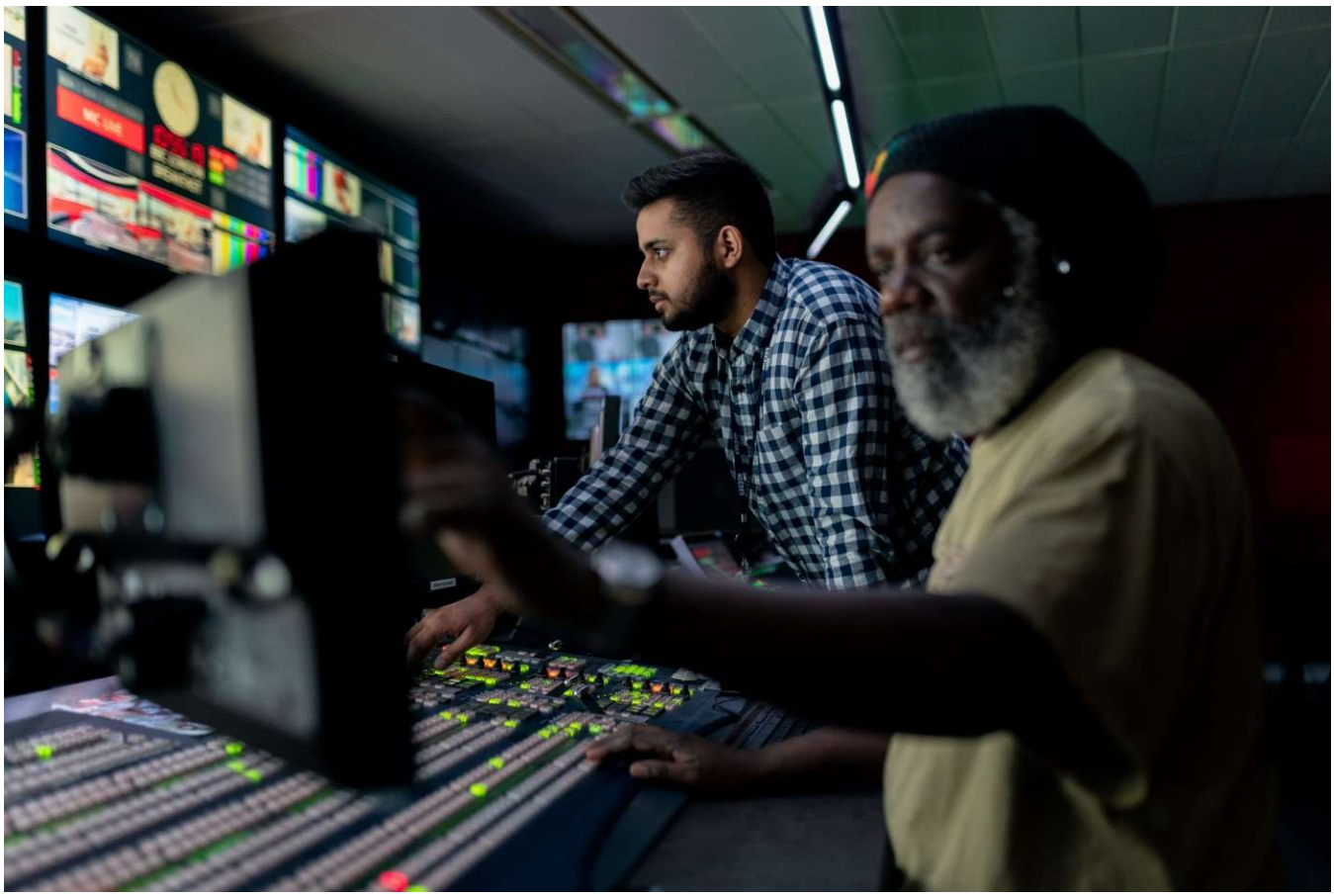


Image by ThisisEngineering RAEng.

An executable version of this model (.pkl) can be saved from the Jupyter notebook:

```
with open('final_model.pkl', 'wb') as f:
    pickle.dump(fitted_models['rf'].best_estimator_, f)
```

The REIT could pre-process new housing data before feeding it into the trained model.

These specialists would r model, ensuring that fresh property data can be pre-processed with our cleaning and feature engineering logic, with predictions pushed downstream to decision-makers on an automated and regular basis.

Final comments

We started with a **business problem**: a company in the business of buying, holding and selling a large portfolio of investment properties wanting to bring consistency and increased performance to their property valuations.

We trained a winning random forest model on a load of historical data comprising over 1,800 past property transactions.

HR can run new data on our trained .pkl file on a manual basis, or an automated pipeline could be built by their engineering department.

Our model was a **regression model**, where the target variable is numerical.

The other side of the coin for supervised learning are **classification models**, whose target variable is categorical. Over here, I trained a binary classification model that predicts whether an employee is at risk of leaving a company.

Finally, I wrote a piece here on where machine learning sits in the field of mathematical modelling.

Follow me on YouTube and Twitter for investment analysis and guides.

My Data Science articles

- Differential Equations versus Machine Learning — here
- Math Modelling versus Machine Learning for COVID-19 — here
- Predict House Prices with Regression — here

- Jupyter Notebooks v

New to Medium?

Join here with my referral link. I will earn a small commission with no extra cost to you. Thanks for your support.

Want to Trade Crypto?

Get a permanent fee discount on the biggest exchanges:

- **FTX**. Get **5% off trading fees** (link)
- **Binance**. Get **5% off trading fees** (link)

Get Free Bitcoin

Sign up below for a free deposit bonus:

- **Nexo**. Get **$25 free BTC** with $100 deposit (link)
- **Celsius**. Get **$50 free BTC** with $400 deposit (link)

Get $25 CRO for FREE

Always wanted a **Crypto.com Metal VISA debit card**?

Which card is right for you?

Get up to 8% cashbacks on daily shopping + free *Spotify*, *Netflix*, *Amazon Prime*, airport lounge access and more!

2. Complete registratic

3. In App, apply for a Ruby Steel card or above

4. Buy and stake $400+ worth of CRO as required

Your shiny new Metal VISA card will arrive in the mail! Woohoo.

…**and** $25 worth of CRO is now instantly unlocked in your Crypto.com App.

Staking CRO provides a host of benefits, such as earning you 8.5% interest on BTC & ETH holdings and 14% on USDC stablecoins!

Thanks for reading!