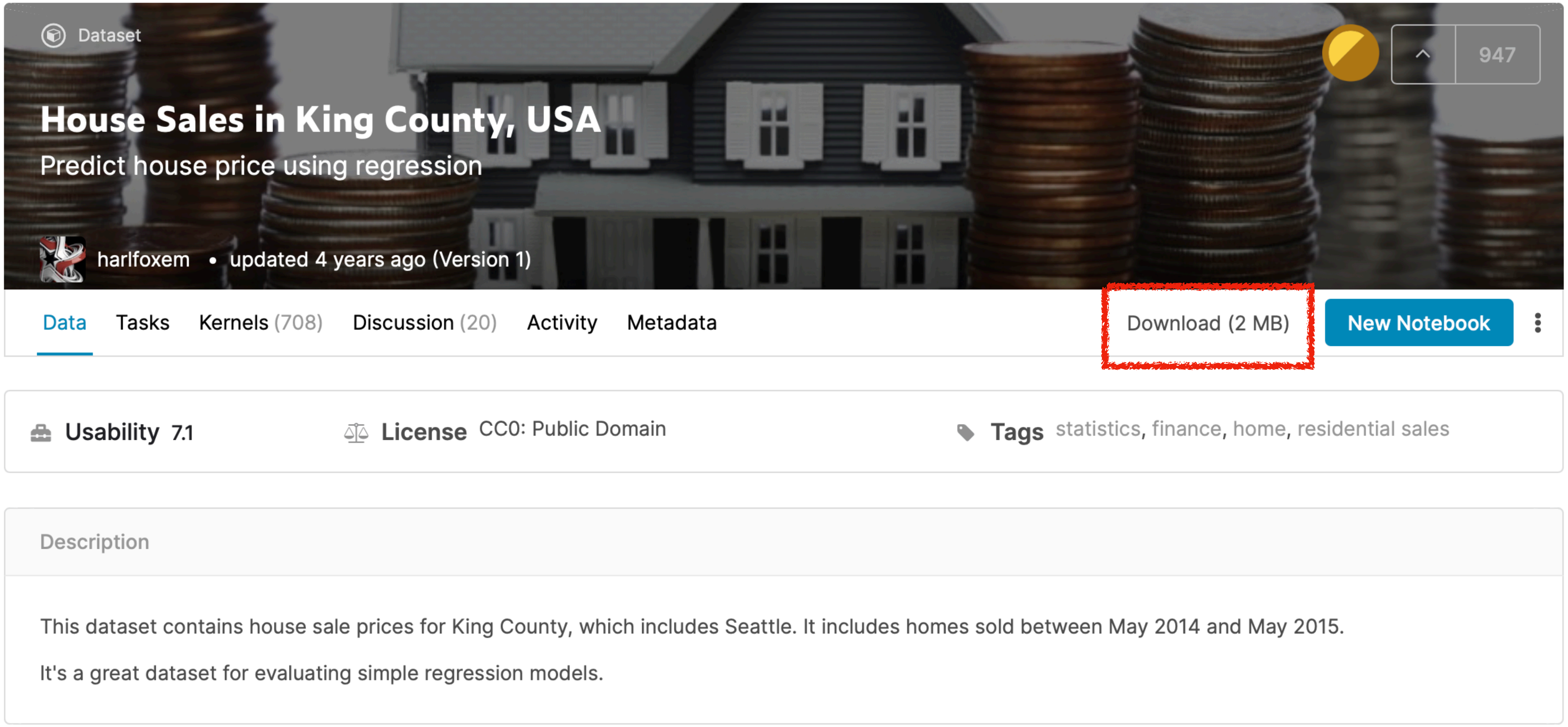
Real world data

Pedro Antonio González Calero

https://www.kaggle.com/harlfoxem/housesalesprediction



House Sales in King County, USA

Predict house price using regression

harlfoxem • updated 4 years ago (Version 1)

Usability 7.1

License CC0: Public Domain

Tags statistics, finance, home, residential sales

Description

This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015.

It's a great dataset for evaluating simple regression models.

There are around 21,000 rows with 20 features. The value we're trying to predict is a floating point number labeled as "price"

Data Visualization & Preprocessing: Pandas

Library Highlights

- A fast and efficient **DataFrame** object for data manipulation with integrated indexing;
- Tools for **reading and writing data** between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format;
- Intelligent **data alignment** and integrated handling of **missing data**: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form;
- Flexible **reshaping** and pivoting of data sets;
- Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets;
- Columns can be inserted and deleted from data structures for **size mutability**;
- Aggregating or transforming data with a powerful **group by** engine allowing split-apply-combine operations on data sets;
- High performance **merging and joining** of data sets;
- **Hierarchical axis indexing** provides an intuitive way of working with high-dimensional data in a lower-dimensional data structure;
- **Time series**-functionality: date range generation and frequency conversion, moving window statistics, date shifting and lagging. Even create domain-specific time offsets and join time series without losing data;
- Highly **optimized for performance**, with critical code paths written in Cython or C.
- Python with *pandas* is in use in a wide variety of **academic and commercial** domains, including Finance, Neuroscience, Economics, Statistics, Advertising, Web Analytics, and more.

https://pandas.pydata.org/docs/getting_started/10min.html

Creating a `Series` by passing a list of values, letting pandas create a default integer index:

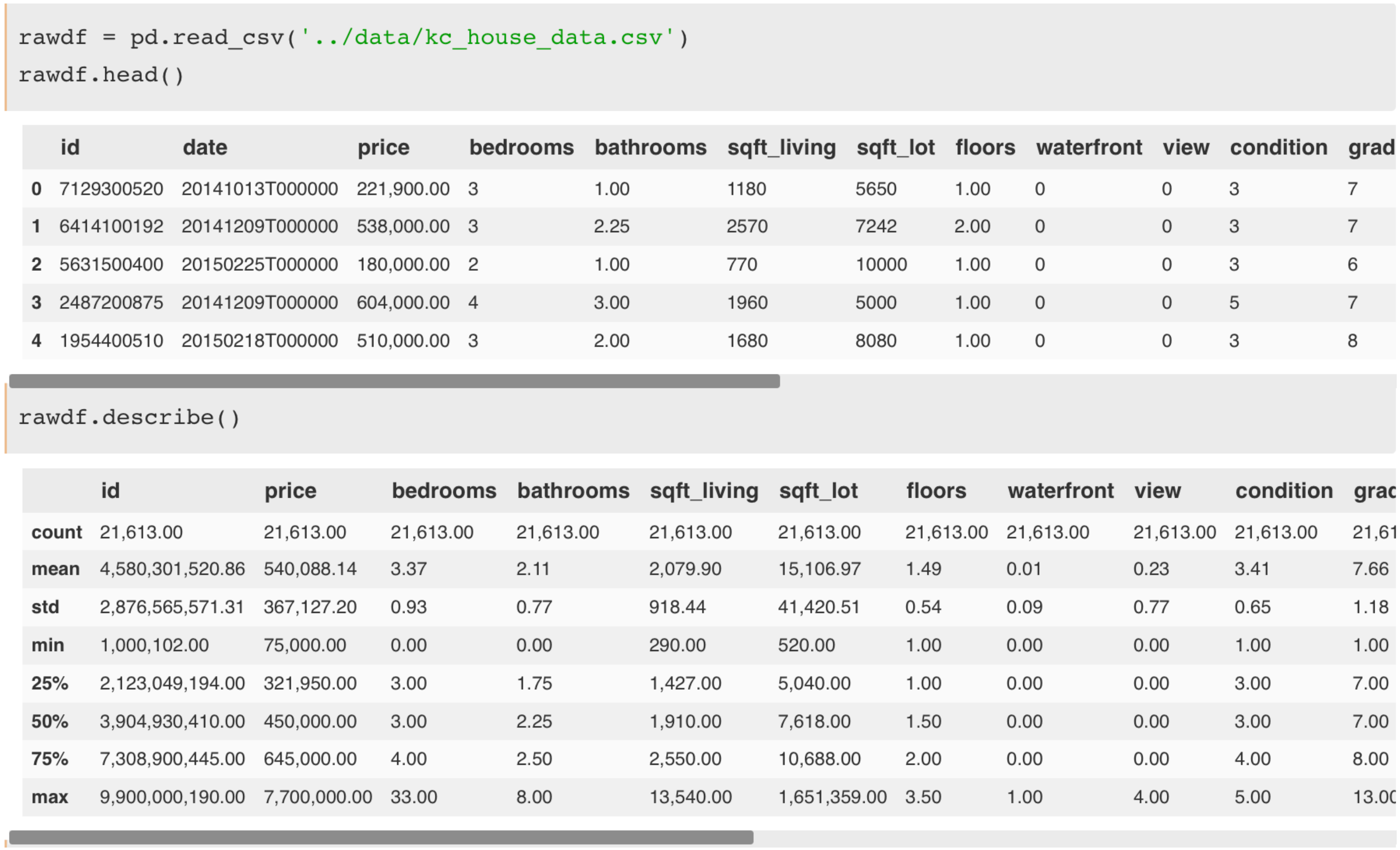
```
In [3]: s = pd.Series([1, 3, 5, np.nan, 6, 8])

In [4]: s
Out[4]:
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Creating a `DataFrame` by passing a NumPy array, with a datetime index and labeled columns:

```
In [5]: dates = pd.date_range('20130101', periods=6)

In [6]: dates
Out[6]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [7]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))

In [8]: df
Out[8]:
                   A         B         C         D
2013-01-01  0.469112 -0.282863 -1.509059 -1.135632
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
2013-01-04  0.721555 -0.706771 -1.039575  0.271860
2013-01-05 -0.424972  0.567020  0.276232 -1.087401
2013-01-06 -0.673690  0.113648 -1.478427  0.524988
```

Data Visualization

Data visualization



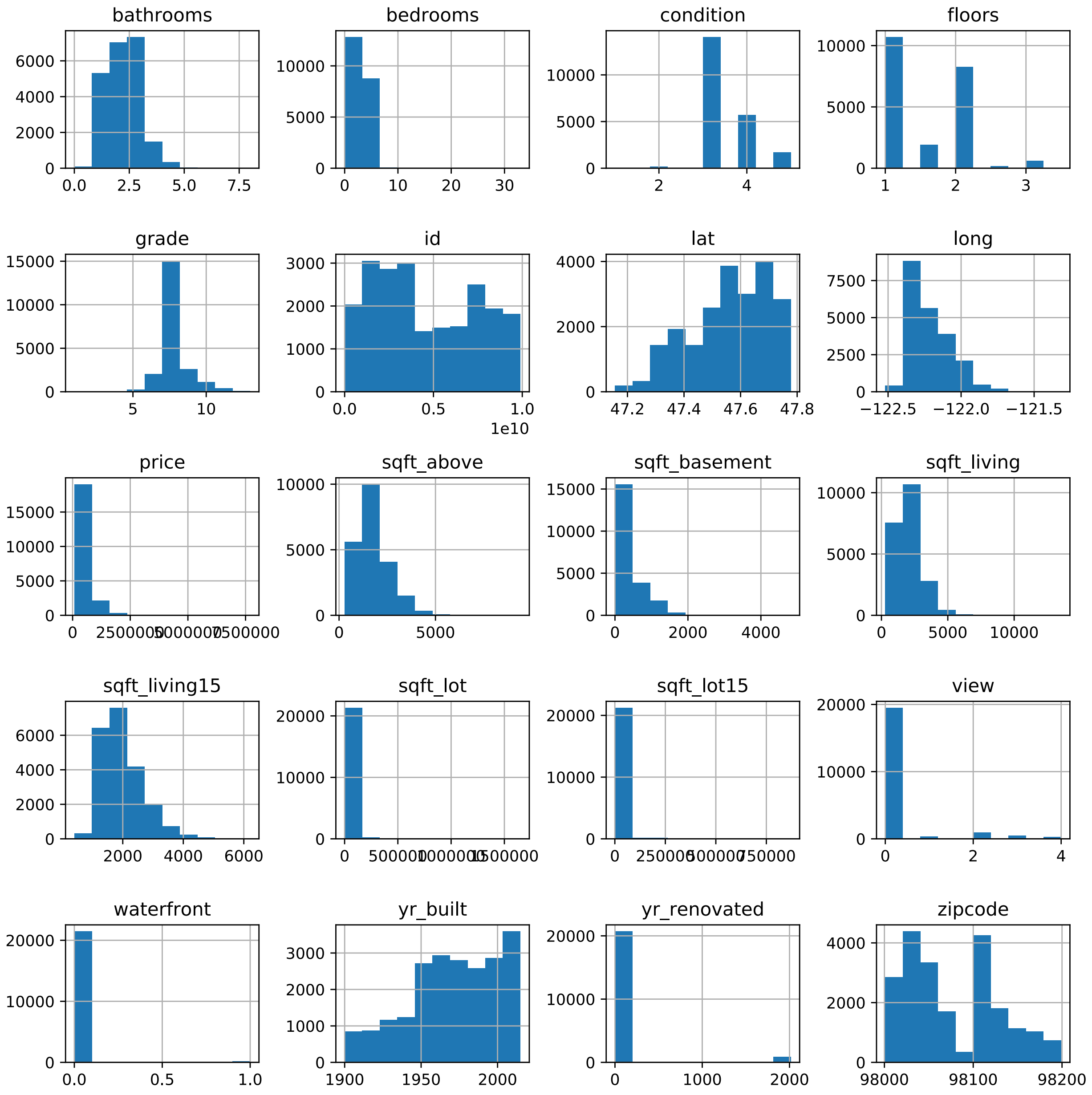
```
rawdf = pd.read_csv('../data/kc_house_data.csv')
rawdf.head()
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grad
0	7129300520	20141013T000000	221,900.00	3	1.00	1180	5650	1.00	0	0	3	7
1	6414100192	20141209T000000	538,000.00	3	2.25	2570	7242	2.00	0	0	3	7
2	5631500400	20150225T000000	180,000.00	2	1.00	770	10000	1.00	0	0	3	6
3	2487200875	20141209T000000	604,000.00	4	3.00	1960	5000	1.00	0	0	5	7
4	1954400510	20150218T000000	510,000.00	3	2.00	1680	8080	1.00	0	0	3	8

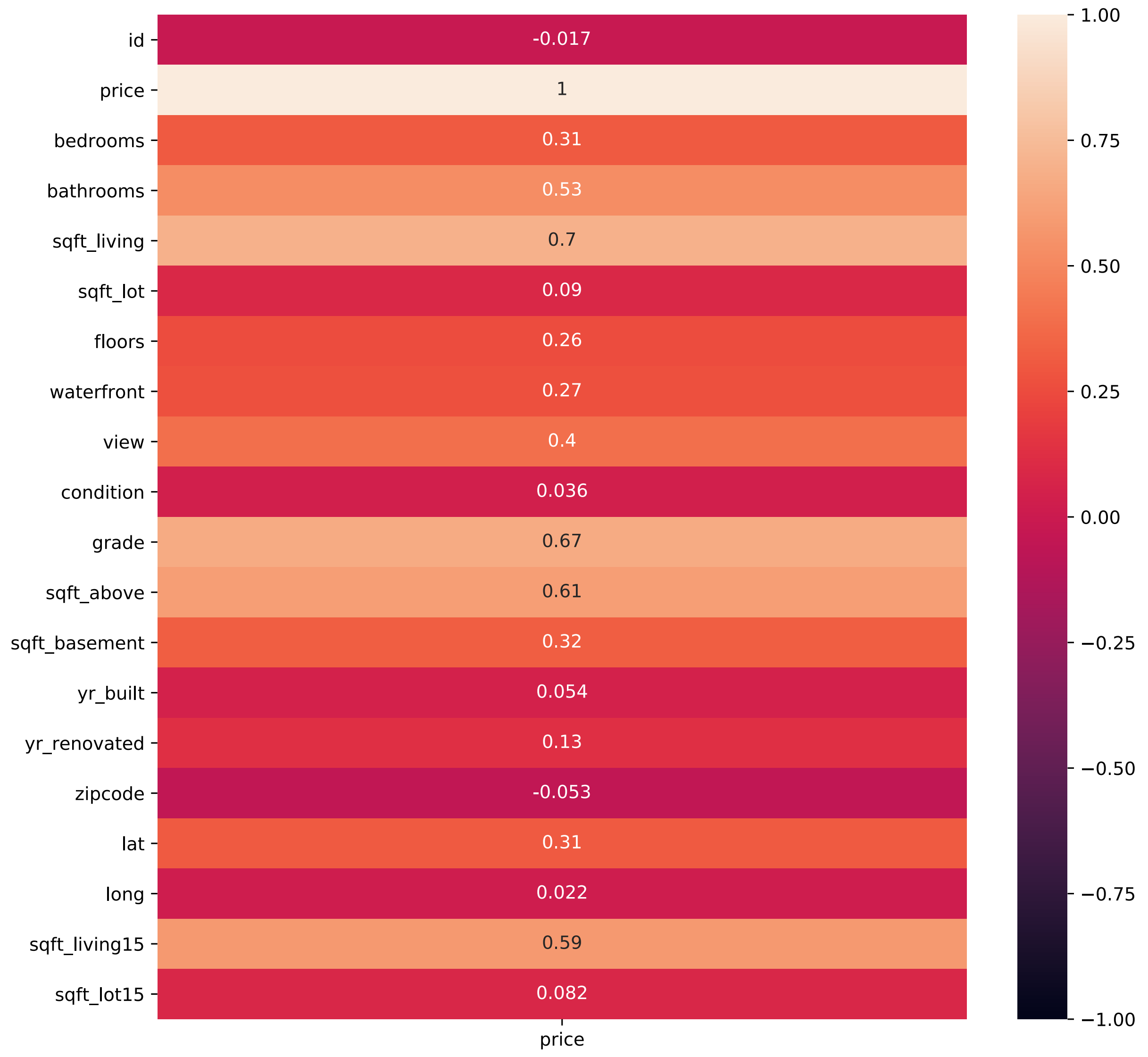
```
rawdf.describe()
```

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grac
count	21,613.00	21,613.00	21,613.00	21,613.00	21,613.00	21,613.00	21,613.00	21,613.00	21,613.00	21,613.00	21,61
mean	4,580,301,520.86	540,088.14	3.37	2.11	2,079.90	15,106.97	1.49	0.01	0.23	3.41	7.66
std	2,876,565,571.31	367,127.20	0.93	0.77	918.44	41,420.51	0.54	0.09	0.77	0.65	1.18
min	1,000,102.00	75,000.00	0.00	0.00	290.00	520.00	1.00	0.00	0.00	1.00	1.00
25%	2,123,049,194.00	321,950.00	3.00	1.75	1,427.00	5,040.00	1.00	0.00	0.00	3.00	7.00
50%	3,904,930,410.00	450,000.00	3.00	2.25	1,910.00	7,618.00	1.50	0.00	0.00	3.00	7.00
75%	7,308,900,445.00	645,000.00	4.00	2.50	2,550.00	10,688.00	2.00	0.00	0.00	4.00	8.00
max	9,900,000,190.00	7,700,000.00	33.00	8.00	13,540.00	1,651,359.00	3.50	1.00	4.00	5.00	13.0

```
rawdf.hist(figsize=(10, 10))
plt.tight_layout()
```



```python
plt.figure(figsize=(6, 8))
sns.heatmap(rawdf.corr()[['price']], annot=True, vmin=-1, vmax=1)
```



https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html

Data preparation

Data preparation

- Standardization, or mean removal and variance scaling

- Encoding categorical features into a new feature of integers: 0 to n_categories - 1.
  Such integer representation can, however, not be used directly with algorithms that expect continuous input, and would interpret the categories as being ordered, which is often not desired (i.e. the set of categories was ordered arbitrarily).
  Another possibility to convert categorical features is to use a one-of-K, also known as one-hot

- Discretization (otherwise known as bucketization or binning) provides a way to partition continuous features into discrete values. Certain datasets with continuous features may benefit from discretization, because discretization can transform the dataset of continuous attributes to one with only nominal attributes.
  One-hot encoded discretized features can make a model more expressive, while maintaining interpretability

- Imputation of missing values
  A basic strategy to use incomplete datasets is to discard entire rows and/or columns containing missing values. However, this comes at the price of losing data which may be valuable (even though incomplete). A better strategy is to impute the missing values, i.e., to infer them from the known part of the data

https://scikit-learn.org/stable/modules/preprocessing.html

Data preparation:

normalization

Normalization

- **Normalization**: Most ML algorithms perform better if the real valued features are scaled to be in a predefined range, for example [0, 1]. This is particularly important for deep neural networks. If the input features consist of large values, deep nets really struggle to learn. The reason is that as the data flows through the layers, with all the multiplications and additions, it gets large very quickly and this negatively affects the optimization process by saturating non-linearities

$$x_i \leftarrow \frac{x_i - \mu_i}{S_i}$$

- **`sklearn.preprocessing.StandardScaler`**
  Standardize features by removing the mean and scaling to unit variance
  Class `StandardScaler` implements the `Transformer` API to compute the mean and standard deviation on a training set so as to be able to later reapply the same transformation on the testing set.

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler(copy=True, with_mean=True, with_std=True)

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> scaler.transform(X_train)
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

The scaler instance can then be used on new data to transform it the same way it did on the training set:

```
>>> X_test = [[-1., 1., 0.]]
>>> scaler.transform(X_test)
array([[-2.44...,  1.22..., -0.26...]])
```

Normalization

```
df = rawdf.copy()

# features that need to be scaled
ss = StandardScaler()
scale_features = ['bathrooms', 'bedrooms', 'grade', 'sqft_above',
                  'sqft_basement', 'sqft_living', 'sqft_living15',
                  'sqft_lot', 'sqft_lot15']
df[scale_features] = ss.fit_transform(df[scale_features])
```

Data preparation:

bucketization

Bucketization

- **Bucketization**: For example the feature which contains the year the house was built (yr_built), ranges from 1900 to 2015. We can categorize it with every year belonging to a distinct category, but then it would be pretty sparse. We would get more signal if we bucketized this feature without losing much information. For example if we use 10 year buckets, years between [1950, 1959] would be collapsed together

- **`pandas.cut`**
  Use `cut` when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable. For example, cut could convert ages to groups of age ranges. Supports binning into an equal number of bins, or a pre-specified array of bins.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]),
...        3, labels=["bad", "medium", "good"])
[bad, good, medium, medium, good, bad]
Categories (3, object): [bad < medium < good]
```

Bucketization

```
bins = range(1890, 2021, 10)
df['yr_built'] = pd.cut(df.yr_built, bins, labels=bins[:-1])

bins = list(range(1930, 2021, 10))
bins = [-10] + bins
df['yr_renovated'] = pd.cut(df.yr_renovated, bins, labels=bins[:-1])

bins = np.arange(47.00, 47.90, 0.05)
df['lat'] = pd.cut(df.lat, bins, labels=bins[:-1])

bins = np.arange(-122.60, -121.10, 0.05)
df['long'] = pd.cut(df.long, bins, labels=bins[:-1])
```

Data preparation:

categorization

Categorization

- **Categorization**: We need to convert categorical data to numerical (NN require numerical data). One-hot representation is used instead of integer encoding when no ordinal relationships exists.
  For example the zip code column contains 4 unique string values: 98039, 98040, 98006 and 98112. After one-hot conversion we will have 4 new binary columns: zipcode_98039, zipcode_98040, zipcode_98006 and zipcode_98112. For a given example, only one of them will have the value 1, the others will be 0

- **`pandas.get_dummies`**
  Convert categorical variable into dummy/indicator variables

```
>>> s = pd.Series(list('abca'))
>>> pd.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

Categorization

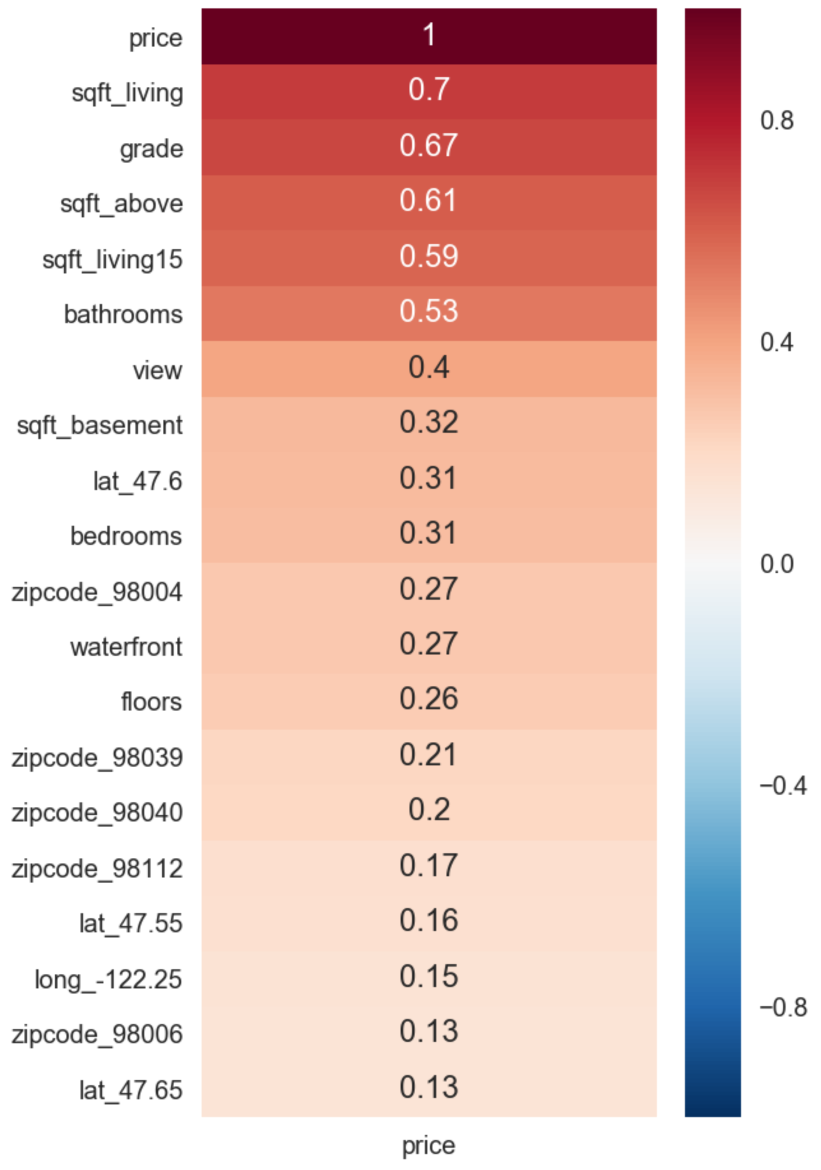
```
# bucketized features
bucketized_features = ['yr_built', 'yr_renovated', 'lat', 'long']

# categorical features
df['date'] = [datetime.strptime(x, '%Y%m%dT000000').strftime(
    '%Y-%m') for x in rawdf['date'].values]
df['zipcode'] = df['zipcode'].astype(np.str)

categorical_features = ['zipcode', 'date']
categorical_features.extend(bucketized_features)

df_cat = pd.get_dummies(df[categorical_features])
df = df.drop(categorical_features, axis=1)
df = pd.concat([df, df_cat], axis=1)
```

https://docs.python.org/3/library/datetime.html

Data Visualization

```
plt.figure(figsize=(10, 10))
tempdf = df.corr()[['price']].sort_values('price', ascending=False).iloc[:20, :]
sns.heatmap(tempdf, annot=True, vmin=-1, vmax=1)
```

After all the transformations we go from 20 to 165 features
The most correlated feature is the square footage, which is expected, bigger houses are usually more expensive. Looking at the list the features make sense. Some zipcodes have high correlation with price, for example 98039 which corresponds to Medina, that's where Bill Gates lives and it's one of the most expensive neighbourhoods in the country



Data preparation:

outliers

Outlier sanitization of the training set

Price value ranges from $75K to $7.7M. A model trying to predict in such a large scale and variance would be very unstable. So we remove outliers and normalize that as well

```
X = df.drop(['price'], axis=1).values
y = df['price'].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=0)

# outlier sanitization of the training set
factor = 5
y_train[np.abs(y_train - y_train.mean()) > (factor * y_train.std())
        ] = y_train.mean() + factor*y_train.std()

# scale the price
ss_price = StandardScaler()

y_train = ss_price.fit_transform(y_train.reshape(-1, 1))
y_test = ss_price.transform(y_test.reshape(-1, 1))
```