King County Dataset
Data Pre-processing and Feature Engineering

This week

Aurélien Géron, ***Hands-on-Machine Learning***

1. Look at the big picture
2. Get the data and set aside a test set
3. Discover and visualise the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Identify a suitable metric for evaluating the task
6. Select a model and train it
7. Fine-tune your model
8. Present your solution
9. Launch, monitor and maintain your system

4. Prepare Data for Machine Learning

- Drop columns with large amounts of missing data
- Data imputation – filling in missing values with various strategies
- Encoding (for categorical data)
- Feature Scaling
- Preparing a ‘pipeline’ of the above processes

4. Dropping columns

- If columns not relevant to investigation or missing too many values to impute, better to remove (drop) these columns.
- Attempting to impute large amounts (e.g. 70-90%) of missing values could lead to artificial patterns that do not represent the true data.
- Rule of thumb suggests dropping columns which have more than 50% of data missing.

- housing.isna().any(axis=1)   //row level
- housing.isna().sum()   //col level
- clean = housing.drop(columns = [“col_name”])

```
1  housing.info()
✓ 0.0s

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 22 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   id               21613 non-null  int64
 1   date             21613 non-null  object
 2   price            21613 non-null  float64
 3   bedrooms         21613 non-null  int64
 4   bathrooms        21613 non-null  float64
 5   sqft_living      21613 non-null  int64
 6   sqft_lot         21613 non-null  int64
 7   floors           21613 non-null  float64
 8   waterfront       21613 non-null  int64
 9   view             21613 non-null  int64
 10  condition        21613 non-null  int64
 11  grade            21613 non-null  int64
 12  sqft_above       21613 non-null  int64
 13  sqft_basement    21613 non-null  int64
 14  yr_built         21613 non-null  int64
 15  yr_renovated     21613 non-null  int64
 16  zipcode          21613 non-null  object
 17  lat              21613 non-null  float64
 18  long             21613 non-null  float64
 19  sqft_living15    21613 non-null  int64
 20  sqft_lot15       21613 non-null  int64
 21  sqft_living_cat  21613 non-null  category
dtypes: category(1), float64(5), int64(14), object(2)
memory usage: 3.5+ MB
```

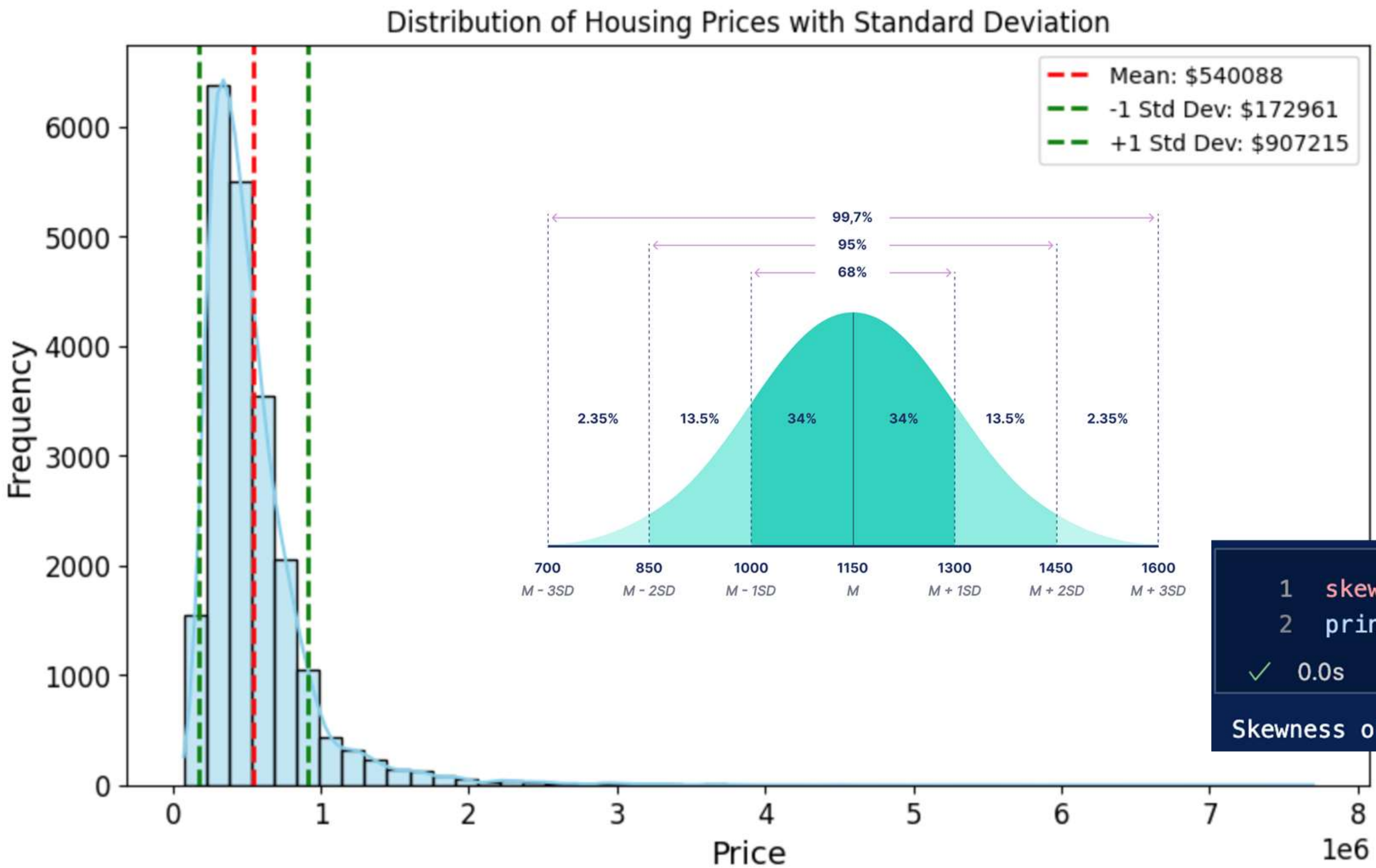
```
1  housing.isna().sum()
✓ 0.0s

id                 0
date               0
price              0
bedrooms           0
bathrooms          0
sqft_living        0
sqft_lot           0
floors             0
waterfront         0
view               0
condition          0
grade              0
sqft_above         0
sqft_basement      0
yr_built           0
yr_renovated       0
zipcode            0
lat                0
long               0
sqft_living15      0
sqft_lot15         0
sqft_living_cat    0
dtype: int64
```

```
1  housing.isnull().any()
✓ 0.0s

id                 False
date               False
price              False
bedrooms           False
bathrooms          False
sqft_living        False
sqft_lot           False
floors             False
waterfront         False
view               False
condition          False
grade              False
sqft_above         False
sqft_basement      False
yr_built           False
yr_renovated       False
zipcode            False
lat                False
long               False
sqft_living15      False
sqft_lot15         False
sqft_living_cat    False
dtype: bool
```

4. Imputation for continuous/categorical data

- Constant Value imputation
  - Replace missing values with a constant (e.g., 0 for numerical, “None” for categorical).
- Numerical data (continuous)
  - Mean imputation **(when normally distributed)**
  - Median imputation **(when data is skewed or significant outliers)**
  - Mode imputation
- Categorical data (classifications)
  - Mode imputation

4. Check distribution of housing data




```
mean_price = housing['price'].mean()
std_price = housing['price'].std()
```

```
skewness = housing['price'].skew()
print(f"Skewness of housing prices: {skewness:.2f}")
```

✓ 0.0s

```
Skewness of housing prices: 4.02
```

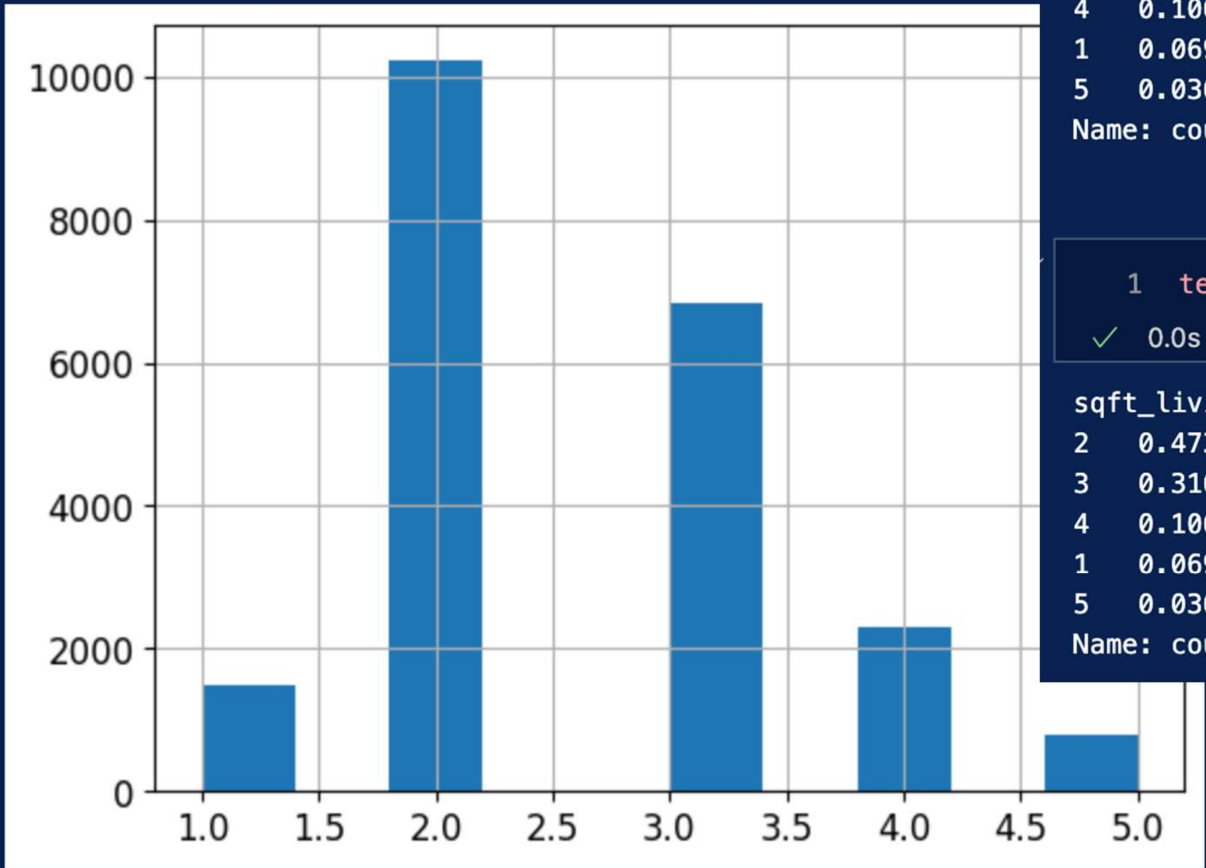
A skewness of 0 indicates a perfect normal distribution.
Positive skewness (> 0): **Right skewed (long tail to the right).**

4. Imputation: global mean vs local mean

- Global mean replaces missing values with the **overall mean of the column**
- Local mean replaces values using the **mean of a subset** (based on categories, clusters or proximity).
  - E.g. if imputing the weight of individuals, calculate the mean weight separately for each age group or gender
  - In our case, we could consider the strata of housing – and impute values accordingly.

```python
housing["sqft_living_cat"] = pd.cut(
    housing.sqft_living,
    bins=[0., 1000., 2000., 3000., 4000., np.inf],
    labels=[1, 2, 3, 4, 5]
)
housing['sqft_living_cat'].hist()
plt.show()
```

✓ 0.0s




```python
train_set.sqft_living_cat.value_counts() / len(train_set)
```

✓ 0.0s

```
sqft_living_cat
2   0.473
3   0.316
4   0.106
1   0.069
5   0.036
Name: count, dtype: float64
```

```python
test_set.sqft_living_cat.value_counts() / len(test_set)
```

✓ 0.0s

```
sqft_living_cat
2   0.473
3   0.316
4   0.106
1   0.069
5   0.036
Name: count, dtype: float64
```

4. Look at Stratas for local means

- 'global' mean for 'sqft_living' col:

```
1  housing['sqft_living'].mean()
✓ 0.0s

2079.8997362698374
```

- Local mean for each 'sqft_living' strata:

```
1  local_means = housing.groupby('sqft_living_cat')['sqft_living'].mean()
2  print(local_means)
✓ 0.0s

sqft_living_cat
1     847.494
2    1524.098
3    2428.103
4    3390.553
5    4803.789
Name: sqft_living, dtype: float64
```

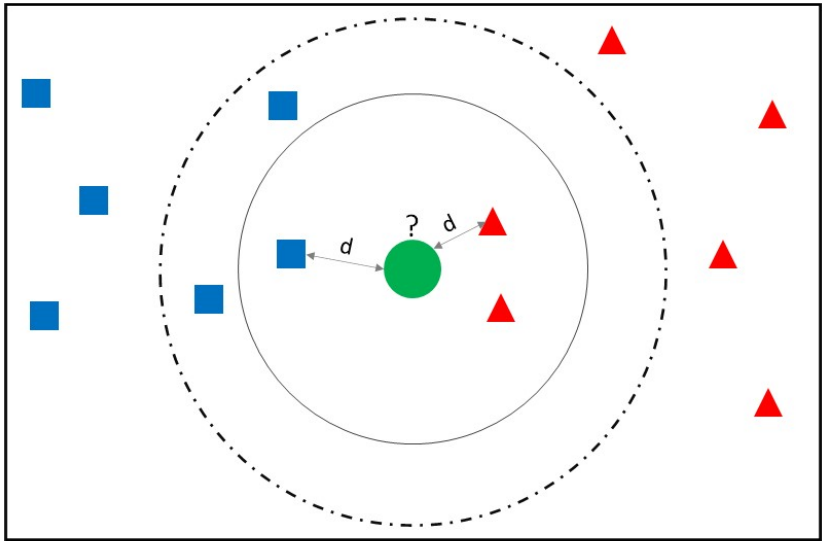
4. Imputing (globally and locally)

- 'global’ imputation for any missing values in a given column:
  - … = df[‘col_name’].**fillna**(df[‘col_name’].**mean()**)
- ‘local’ imputation for any missing values in a given column:
  - … = df['col_name'].**fillna(**df['col_name'].**map**(**local_means**))

4. Other types of imputation

- K-NN Imputation can estimate missing values based on the mean or median of the k nearest neighbours.

```python
from sklearn.impute import KNNImputer

imputer = KNNImputer(n_neighbors=5)
housing_imputed = imputer.fit_transform(housing)
```



- Can also apply to select columns of a dataset too

4. Sklearn's SimpleImputer for continuous

- Sklearn's SimpleImputer class can use different a 'Strategy' such as 'mean', 'median', 'most_frequent', or 'constant'
- Sklearn produces an object which can then be '**fit()**' to the training data
- We then call the '**transform()**' function to apply the 'learnt' values to the missing fields.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="most_frequent")
train_data_num = train_data.select_dtypes(include=[np.number])
imputer.fit(train_data_num)
train_data_arr = imputer.transform(train_data_num)
```

4. RobustImputer

- The third party 'Dirty cat' RobustImputer class uses **robust statistics** such as the **median** and **interquartile range** (IQR) to impute missing values.
- By focusing on the median and IQR, it is less sensitive to outliers, which makes it a good choice when your data contains extreme values or outliers that might skew the mean.

```
from dirty_cat import RobustImputer

imputer = RobustImputer() #IQR by default
housing['bedrooms'] = imputer.fit_transform(housing[['bedrooms']])
```

4. Encoding for categorical data

- Categorical data
  - Ordinal – with rank order (grades: ‘A’, ‘B’, ‘C’)
  - Nominal – no order (colors: ‘blue’, ‘green’, ‘red’)

- To encode is to translate –from text to numeric

- Ordinal encoding (preserve rank order)
- One Hot encoding (for nominal / no ranking)

- Can also impute missing values with the mode (most frequent)

4. Encoding for categories

- Sklearn.preprocessing.OrdinalEncoder can be used for assigning string values of columns to numerical categories (integers)
- Importantly, rank order is also preserved

Original Data:

Color	Size	Price
Blue	L	100
Green	M	150
Red	S	200
Green	XL	120
Red	M	180

Label Encoding

Label Encoded Data:

Color	Size	Price
0	0	100
1	1	150
2	2	200
1	3	120
2	1	180

```python
from sklearn.preprocessing import OrdinalEncoder
condition_order = ['bad', 'average', 'good', 'very good', 'excellent']
encoder_condition = OrdinalEncoder(categories=[condition_order])
housing['condition_encoded'] = encoder_condition.fit_transform(housing[['condition']])
```

4. Encoding for categories

- Sklearn.preprocessing.OneHotEncoding
- Create a binary attribute per category (additional columns for each row)
- This category is turned **on** (1 – hot) or **off** (0 – cold) depending on whether this value is present for the row.

id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

```python
zg_copy = train_data[['zipcode_group']]

cat_encoder = OneHotEncoder(categories='auto')
train_data_cat = train_data[["zipcode_group"]]
train_data_cat_1hot = cat_encoder.fit_transform(train_data_cat)
train_data_cat_1hot
```

```
<17290x9 sparse matrix of type '<class 'numpy.float64'>'
	with 17290 stored elements in Compressed Sparse Row format>
```

```python
cat_encoder.categories_
```

```
[array(['zg_0', 'zg_1', 'zg_2', 'zg_3', 'zg_4', 'zg_5', 'zg_6', 'zg_7',
        'zg_8'], dtype=object)]
```

```python
train_data_cat_1hot.toarray()
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 1., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```python
train_data_cat_1hot.toarray().sh
```

```
(17290, 9)
```

4. Feature Scaling

- For normalisation of values and also to reduce range
- MinMaxScaler
- StandardScaler
- There are two common ways to get all attributes to have the same scale:
- min-max scaling: rescaling the range of features to scale the range in [0, 1] or [−1, 1] (using scikit-learn MinMaxScaler)
- standardisation: scales the data around the **mean of 0** and **variance = 1** (using scikit-learn StandardScaler).
  - Not between -1 and +1

4. Feature Scaling

```python
from sklearn.preprocessing import MinMaxScaler

ages = [[30], [40], [50], [60]]
scaler = MinMaxScaler()
scaled_ages = scaler.fit_transform(ages)

print("Original ages:")
print(ages)
print("\nScaled ages (min-max scaled):")
print(scaled_ages)
```

```
Original ages:
[[30], [40], [50], [60]]

Scaled ages (min-max scaled):
[[0.  ]
 [0.33]
 [0.67]
 [1.  ]]
```

```python
from sklearn.preprocessing import StandardScaler

ages = [[10], [21], [11], [10]]
scaler = StandardScaler()
standardized_ages = scaler.fit_transform(ages)

print("Original ages:")
print(ages)
print("\nStandardized ages:")
print(standardized_ages)
```

```
Original ages:
[[10], [21], [11], [10]]

Standardized ages:
[[-0.65]
 [ 1.73]
 [-0.43]
 [-0.65]]
```

4. Bringing all this together with Pipeline

- Think of a literal pipeline
- We can create a ‘wrapper’ for these different steps:
  - Dropping columns
  - Imputation
  - Scaling

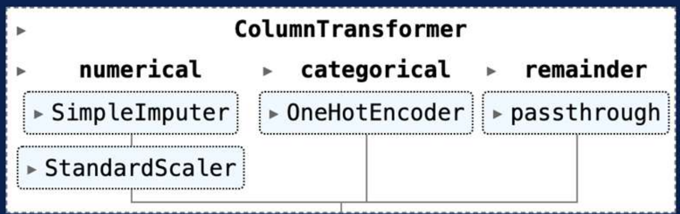
```
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('std_scaler', StandardScaler())
])
train_data_num_scaled = num_pipeline.fit_transform(train_data[num_feats])
```

4. Bringing all this together with Column Transformation

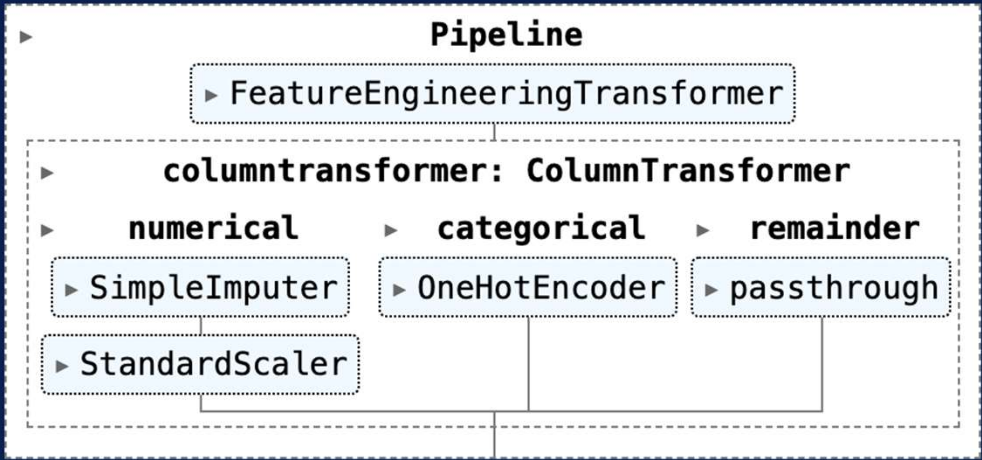
- Furthermore, Column Transformation can handle both continuous and categorical feature engineering:

```python
from sklearn.compose import ColumnTransformer

column_transformer = ColumnTransformer(
    (
        ("numerical", num_pipeline, num_feats),
        ("categorical", OneHotEncoder(categories='auto', sparse_output=False).set_output(transform="pandas"), cat_feats),
    ),
    remainder="passthrough",
    verbose_feature_names_out=False,
).set_output(transform="pandas")

column_transformer
```

Python



```python
full_pipeline.fit(train_set.drop(columns=["price"]), train_set["price"])
```




```python
train_data_prepared =  full_pipeline.transform(train_set.drop(columns=["price"]))
train_data_prepared
```

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	view	condition	grade	sqft_lot15
20474	-0.409	1.479	-0.767	-0.330	2.794	-0.305	-0.629	0.290	-0.427
3840	-1.509	-1.455	-1.380	-0.108	-0.915	-0.305	0.910	-0.557	-0.054
7426	-0.409	1.805	2.367	0.159	0.940	-0.305	-0.629	1.985	0.143
4038	0.691	-1.455	-1.030	-0.209	0.013	-0.305	-0.629	-1.405	-0.424

Summary – Preprocessing steps

- Drop columns with large amounts of missing data (> 50%)
- Data imputation – filling in missing values with various strategies
- Encoding (for categorical data)
- Feature Scaling
- Preparing a ‘pipeline’ of the above processes

Next week

Aurélien Géron, ***Hands-on-Machine Learning***

1. Look at the big picture
2. Get the data and set aside a test set
3. Discover and visualise the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Identify a suitable metric for evaluating the task
6. Select a model and train it
7. Fine-tune your model
8. Present your solution
9. Launch, monitor and maintain your system