King County Dataset

Regression, SGD, Evaluation Metrics

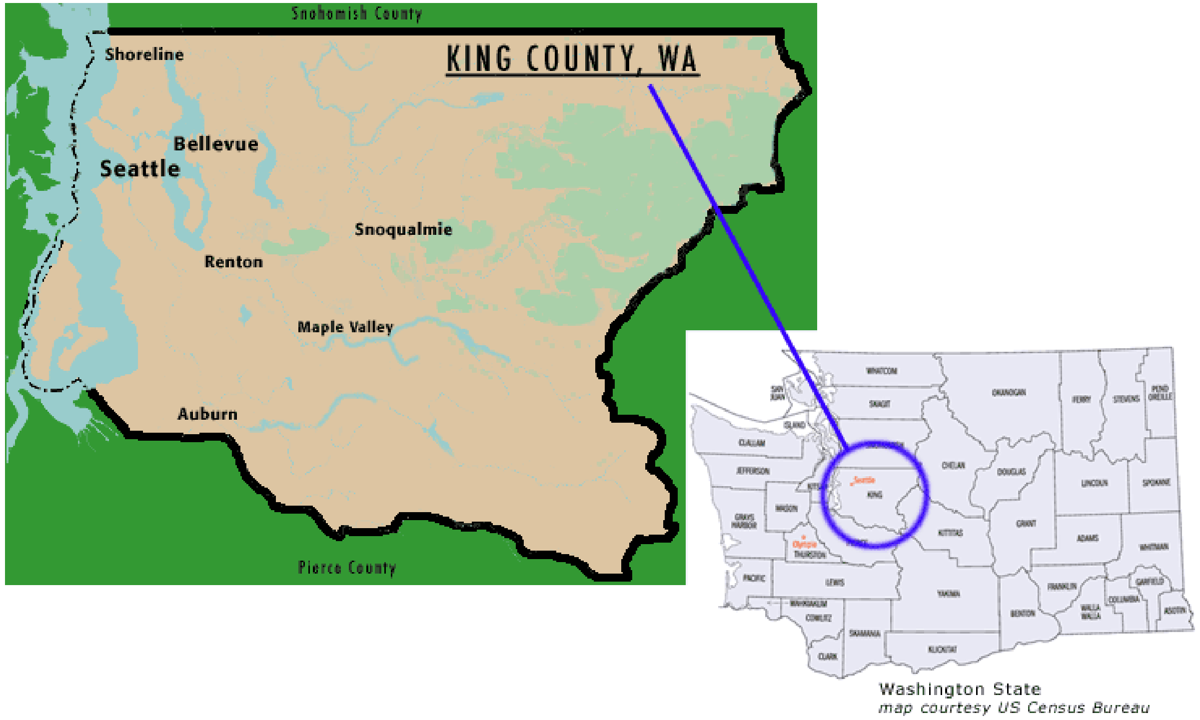
Last Week

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

1. Frame the Problem

- We want to be able to **predict the price of houses** in King County, Washington, US.



- Questions for you to consider:
  - Is it **supervised**, unsupervised, or reinforcement learning?
  - Is it a classification task, a **regression task** or something else?
  - Should you use batch learning or online learning techniques?

2. Get the data

First of all let's import the data from the CSV file.

```python
housing = pd.read_csv("../datasets/kings-county-housing-data.csv")
```

kings-county-housing-data

id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	lat	long	sqft_living15	sqft_lot
7129300520	20141013T000000	221900.0	3	1.0	1180	5650	1.0	0	0	3	7	1180	0	1955	0	47.5112	-122.257	1340	56
6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	3	7	2170	400	1951	1991	47.721	-122.319	1690	76
5631500400	20150225T000000	180000.0	2	1.0	770	10000	1.0	0	0	3	6	770	0	1933	0	47.7379	-122.233	2720	80
2487200875	20141209T000000	604000.0	4	3.0	1960	5000	1.0	0	0	5	7	1050	910	1965	0	47.5208	-122.393	1360	50
1954400510	20150218T000000	510000.0	3	2.0	1680	8080	1.0	0	0	3	8	1680	0	1987	0	47.6168	-122.045	1800	75
7237550310	20140512T000000	1225000.0	4	4.5	5420	101930	1.0	0	0	3	11	3890	1530	2001	0	47.6561	-122.005	4760	1019
1321400060	20140627T000000	257500.0	3	2.25	1715	6819	2.0	0	0	3	7	1715	0	1995	0	47.3097	-122.327	2238	68
2008000270	20150115T000000	291850.0	3	1.5	1060	9711	1.0	0	0	3	7	1060	0	1963	0	47.4095	-122.315	1650	97
2414600126	20150415T000000	229500.0	3	1.0	1780	7470	1.0	0	0	3	7	1050	730	1960	0	47.5123	-122.337	1780	81
3793500160	20150312T000000	323000.0	3	2.5	1890	6560	2.0	0	0	3	7	1890	0	2003	0	47.3684	-122.031	2390	75
1736800520	20150403T000000	662500.0	3	2.5	3560	9796	1.0	0	0	3	8	1860	1700	1965	0	47.6007	-122.145	2210	89
9212900260	20140527T000000	468000.0	2	1.0	1160	6000	1.0	0	0	4	7	860	300	1942	0	47.69	-122.292	1330	60
114101516	20140528T000000	310000.0	3	1.0	1430	19901	1.5	0	0	4	7	1430	0	1927	0	47.7558	-122.229	1780	126
6054650070	20141007T000000	400000.0	3	1.75	1370	9680	1.0	0	0	4	7	1370	0	1977	0	47.6127	-122.045	1370	102

2. stratified_split ahead of EDA

```
from sklearn.model_selection import StratifiedShuffleSplit
splitter = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in splitter.split(housing, housing.sqft_living_cat):
    train_set = housing.loc[train_index]
    test_set = housing.loc[test_index]
```

✓ 0.0s

```
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
```

```
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
```

3. Inspect the data

Description of the features:

Here follows a detailed description of all the features (i.e. columns/variables) in the dataset.

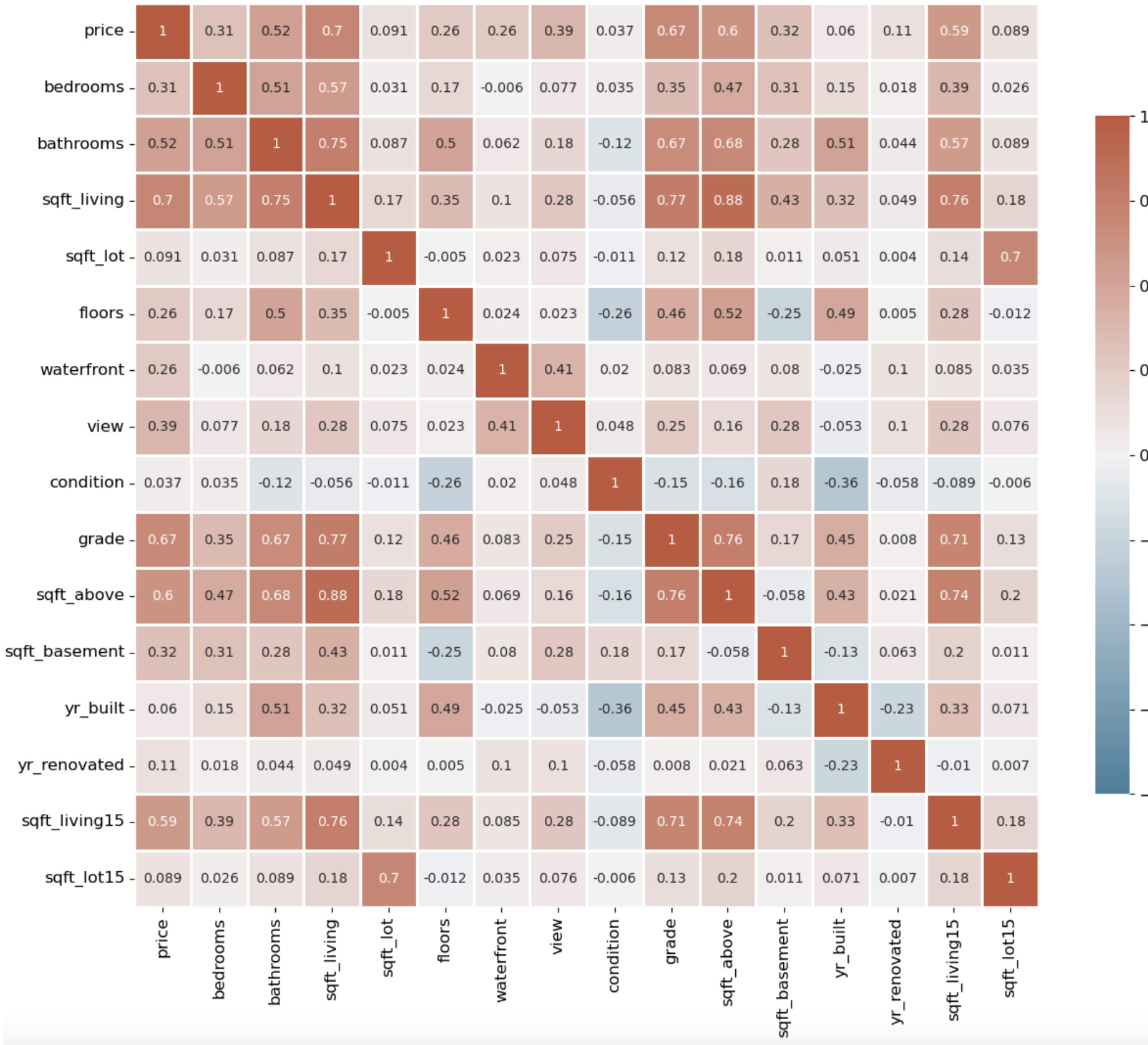
- **id** - unique identifier for a house
- **date** - house was sold
- **price** - price, our prediction target
- **bedrooms** - number of Bedrooms/House
- **bathrooms** - number of bedrooms
- **sqft_living** - square footage of the home
- **sqft_lot** - square footage of the entire lot
- **floors** - total number of floors (levels) in house
- **waterfront** - house which has a view to a waterfront
- **view** - quality of view
- **condition** - how good the condition is (overall)
- **grade** - overall grade given to the housing unit, based on King County grading system
- **sqft_above** - square footage of house apart from basement
- **sqft_basement** - square footage of the basement
- **yr_built** - Built Year
- **yr_renovated** - Year when house was renovated
- **zipcode_group** - 9 groups aggregating some of the 70 zipcodes having similar characteristics
- **lat** - Latitude coordinate
- **long** - Longitude coordinate
- **sqft_living15** - The square footage of interior housing living space for the nearest 15 neighbours
- **sqft_lot15** - The square footage of the land lots of the nearest 15 neighbours

```
housing.info()
```

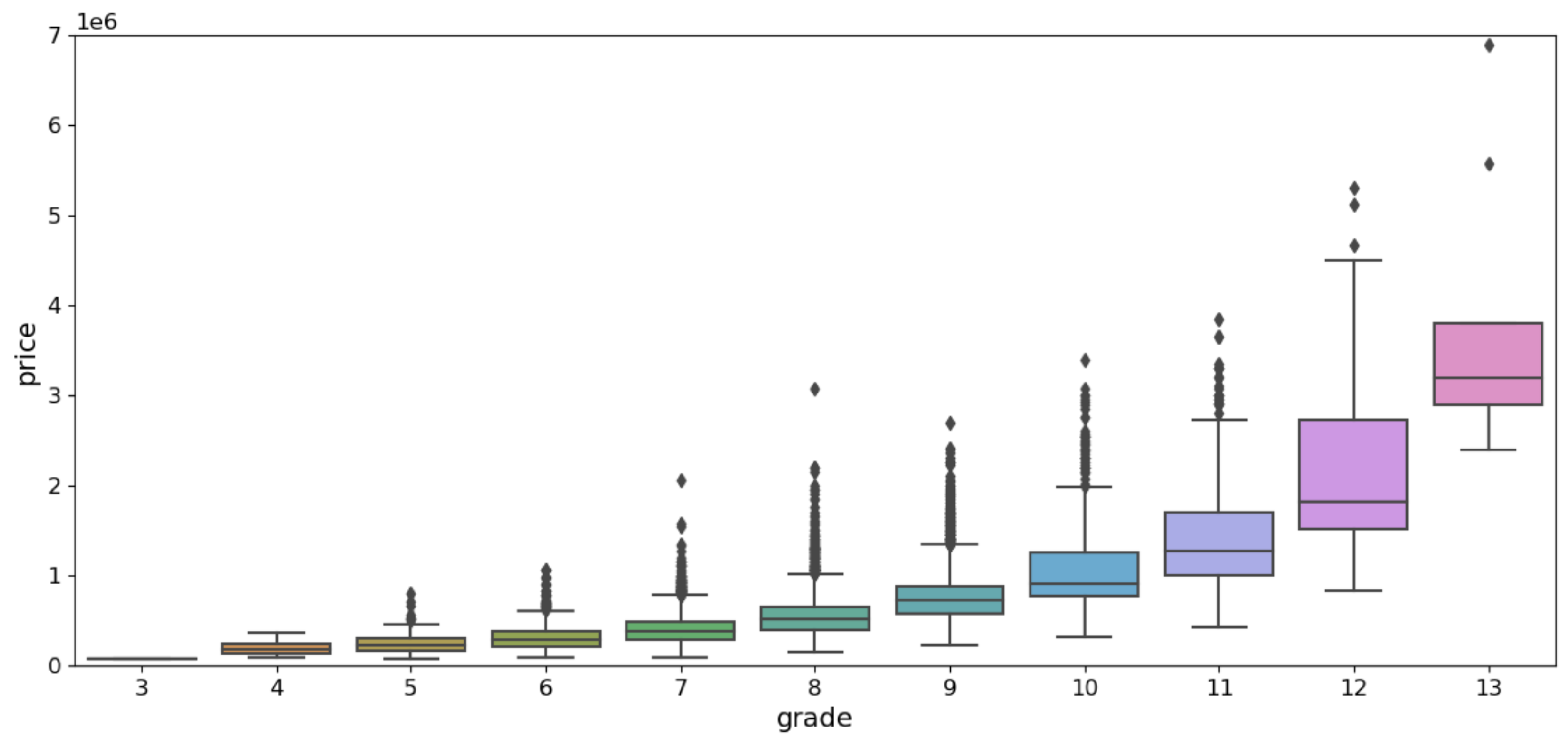
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21613 non-null  int64
 1   date           21613 non-null  object
 2   price          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  lat            21613 non-null  float64
 17  long           21613 non-null  float64
 18  sqft_living15  21613 non-null  int64
 19  sqft_lot15     21613 non-null  int64
 20  zipcode_group  21613 non-null  object
dtypes: float64(5), int64(14), object(2)
memory usage: 3.5+ MB
```

sns.heatmap

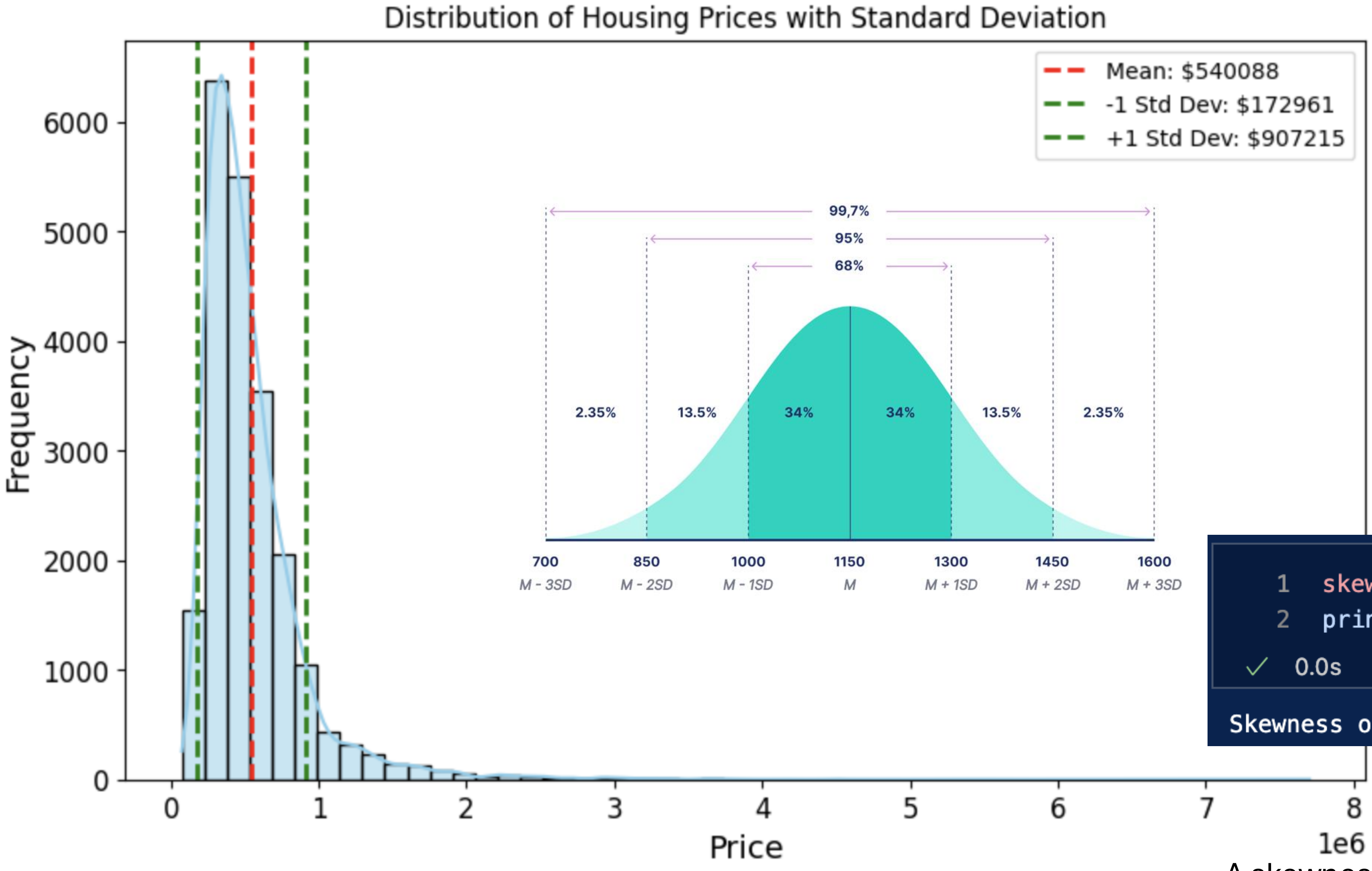
- Pearson's correlation coefficient drawn as a 'heat' map
- Colour coded for accessibility
- +1 = perfect positive correlation – hotter!
- -1 = perfect negative correlation – cooler!



sns.boxplot



3. Check distribution of housing data




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

```python
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. Prepare the data (pre-processing / cleaning)

- Drop columns with large amounts of missing data (> 50%)
- Data imputation – filling in missing values with various strategies
  - Mean (when normally distributed)
  - Median (when data is skewed or significant outliers)
  - Mode (for categorical data)
- Encoding (for categorical data)
  - Ordinal (for preserving order / ranking)
  - One Hot (creates a column for each category)
- Feature Scaling
  - Min Max Scaler [-1 to +1] or [0 to +1]
  - Standard Scaler (scaled around the **mean of 0** and **variance = 1)**
- Preparing a ‘pipeline’ of the above processes

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

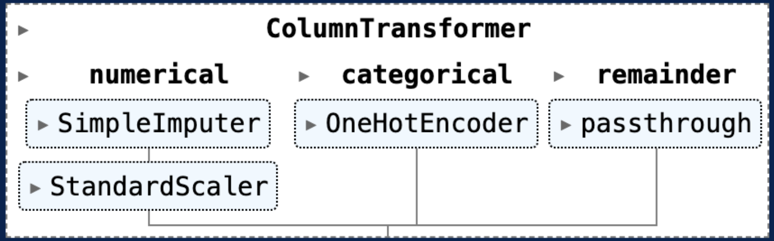
```
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('std_scaler', StandardScaler())
])
train_data_num_scaled = num_pipeline.fit_transform(train_data[num_feats])
```

4. Bringing all this together with Column Transformation

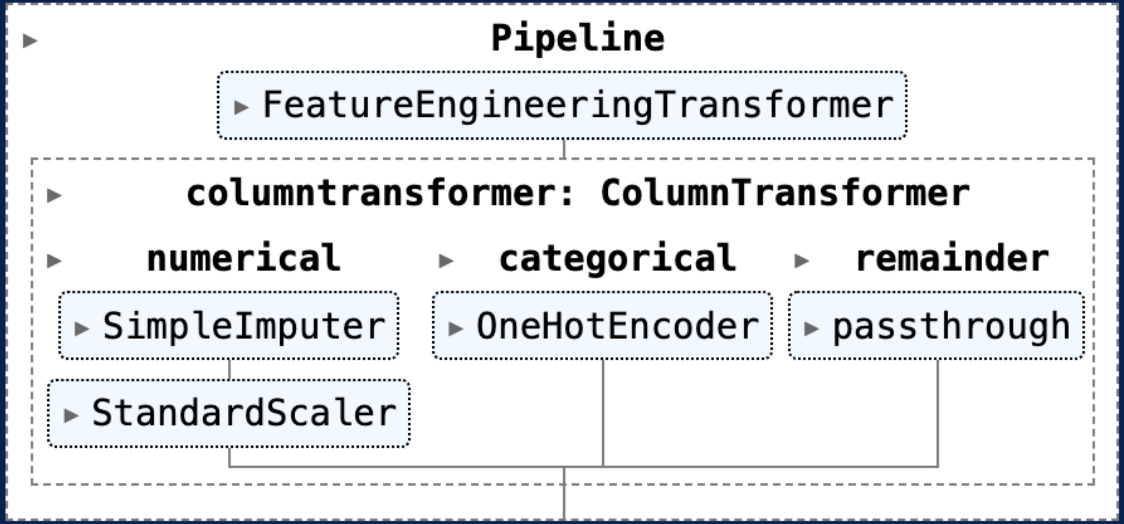
- Furthermore, Column Transformation can handle both continuous and categorical feature engineering:

```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

This Week

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

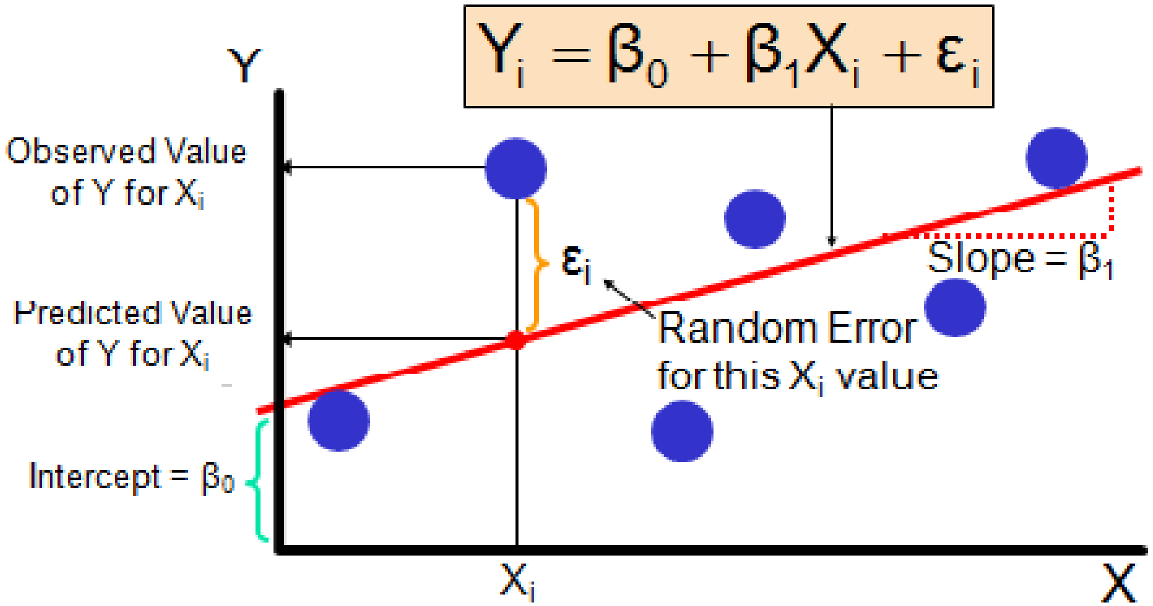
1. Look at the big picture
2. Get the data and set aside a test set
3. Discover and visualise the data to gain insights
4. Prepare the data for Machine Learning algorithms
5. Identify a suitable metric for evaluating the task
6. Select a model and train it
7. Fine-tune your model
8. Present your solution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

Agenda

- Evaluation Metrics for Regression Tasks: MAE, MSE, RMSE, R^2
- Regression models – Linear to Polynomial
- Gradient Descent algo – for the SGDRegressor
- Hyperparameters vs parameters (learnt by the model)
- Regularisation and Lasso / Ridge penalties
- Cross-Validation (CV)

5. Identify a metric for evaluation

- **Metrics for Regression tasks**:
  - Mean Squared Error (MSE)
  - Root Mean Squared Error (RMSE)
  - Mean Absolute Error (MAE)
  - R^2 (variance explanation)



- **Metrics for Classification tasks**:
  - Precision -> F1-score
  - Recall
  - Accuracy
  - Confusion Matrix (type I and II errors)

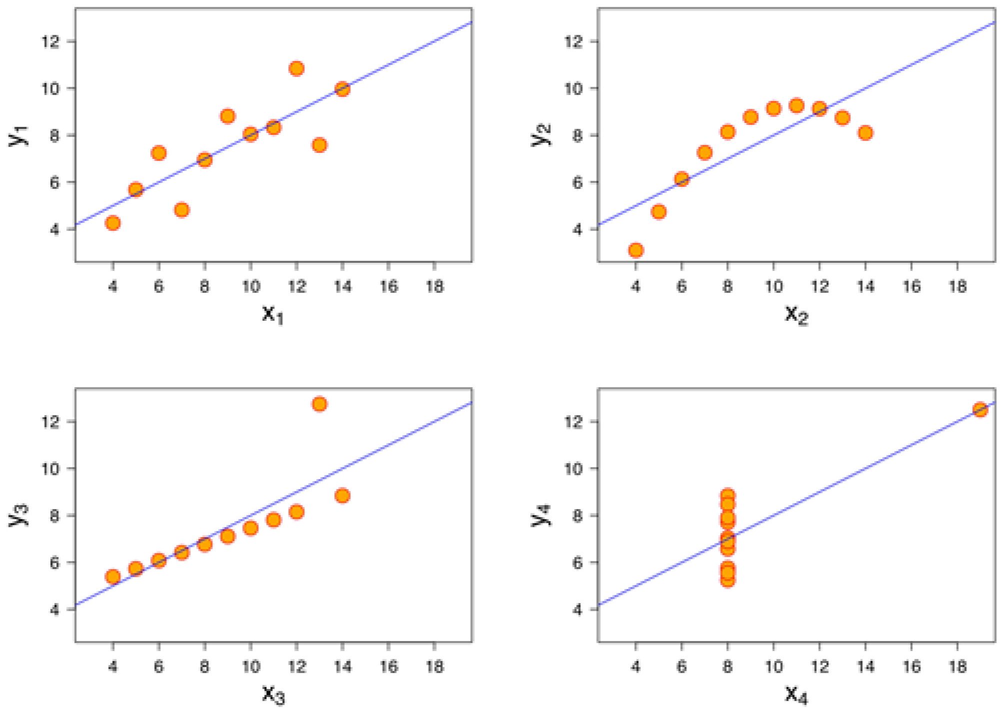
Predicted

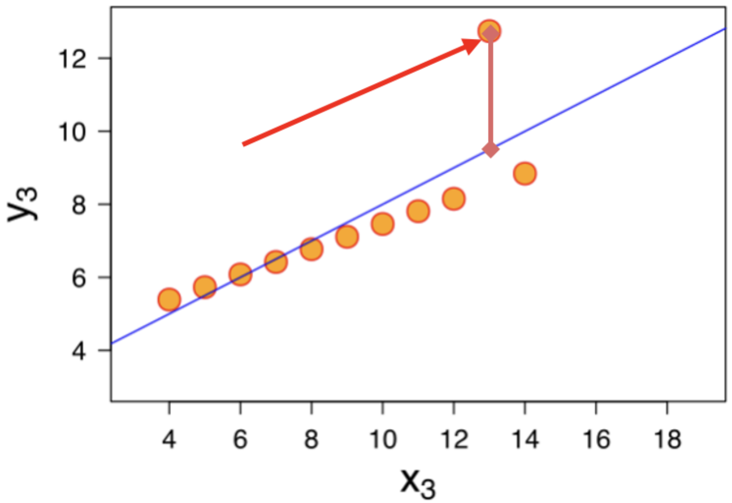
Actual	Spam	Non-spam
Spam	**600** (True positive)	**300** (False negative)
Non-spam	**100** (False positive)	**9000** (True negative)

5. Metrics: Mean Absolute Error (MAE)

$$\frac{1}{N}\sum_{i=1}^{N}|y_i - \widehat{y_i}|$$

- This is the mean of the absolute value of errors.
- Abs(7 - 5) = 2
- Abs(5 – 7) = 2

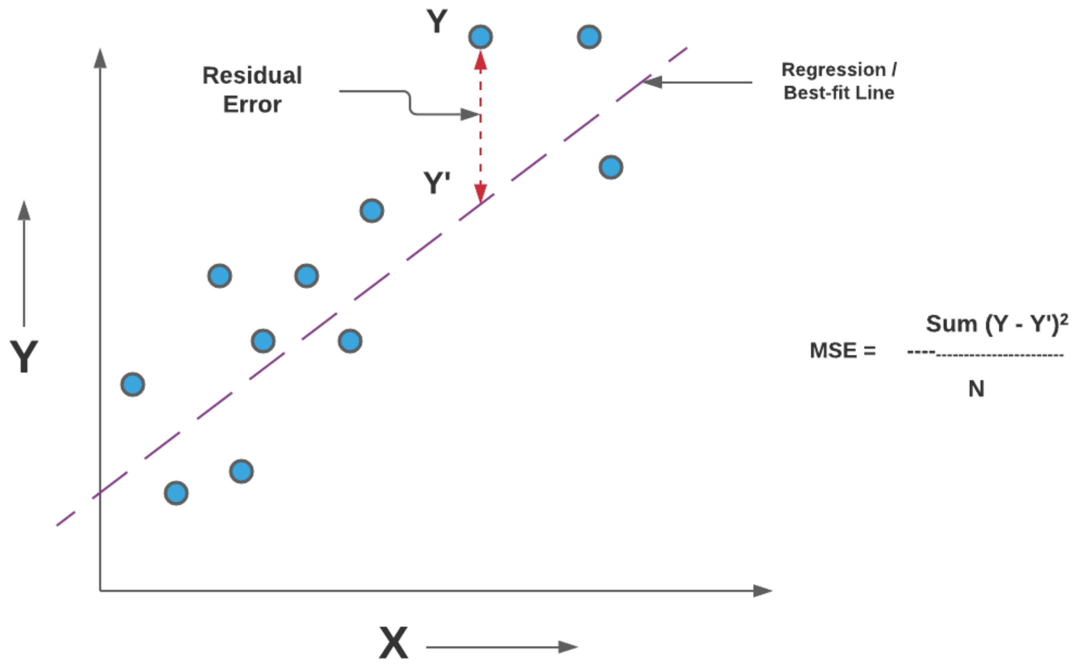




5. Metrics: Mean Squared Error (MSE)

- This is the mean of the squared errors.
- Larger errors are noted more than with MAE, making MSE more popular.

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$



5. Metrics: Root Mean Squared Error (RMSE)

- Root Mean Squared Error (RMSE)
- This is the root of the mean of the squared errors.

$$\sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2}$$

- Most popular as it has same units as the **target variable (y)**
  - Easier to interpret!
- Context matters: A RMSE of $10 is fantastic for predicting our house prices, but not so good for predicting the price of coffee!

5. Metrics: R^2

$$R^2 = 1 - \frac{MSE(model)}{MSE(baseline)} = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

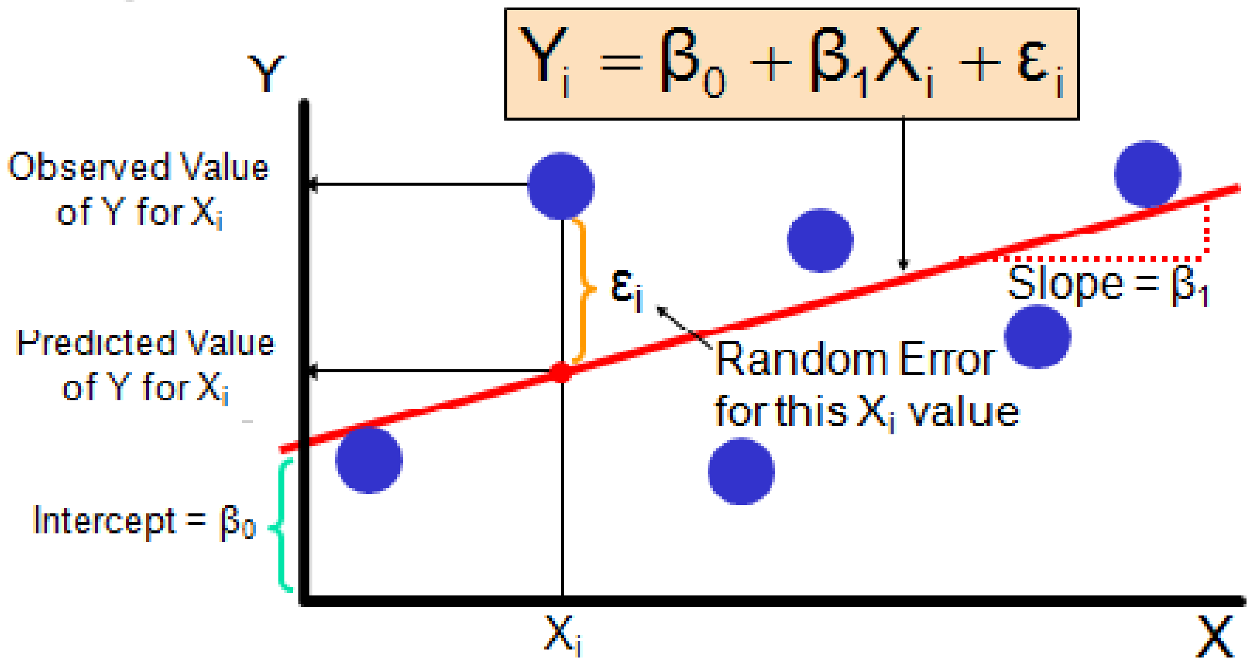
- The R^2 **score** (also called the **coefficient of determination**) measures how well the regression model explains the variance in the target variable.
- **The numerator** represents the **unexplained variance** (errors in prediction).
- **The denominator** represents the **total variance** in y_test.
- **1.0** Perfect fit (model explains all variance)
- **0.0** Model explains **none** of the variance
- **Negative** Model performs **worse than a horizontal line**

6. Select a model (algorithm) and train it

- Linear Regression: Ordinary Least Squares
  - Closed form solution (Normal Equation)
  - Gradient Descent
- Polynomial Regression
- Regularized Models
  - Ridge Regression
  - Lasso Regression
- Decision Trees Regression
- Something else (Support Vector Machines, Neural Networks…)
- Ensemble Models, Random Forests

6. Linear Regression

- Find the best linear model that fits our data
- This means finding two parameters: **slope** (β1) and **intercept** (β0)



Once trained, we can use the model to make predictions => machine learning!!

6. Linear Regression

- Find the best linear model that fits our data
- This means finding two parameters: **slope** (β1) and **intercept** (β0)

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train, y_train)
```

✓ 0.0s

```
▾ LinearRegression
LinearRegression()
```

6. Linear Regression: sqft_living vs price

- Let's look at sqft_living vs price:

```python
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train[['sqft_living']], y_train)
```

✓ 0.0s

```
▾ LinearRegression ℹ ?
LinearRegression()
```

```python
print("Slope (Coefficient):", model.coef_)
print("Intercept:", model.intercept_)
```

✓ 0.0s

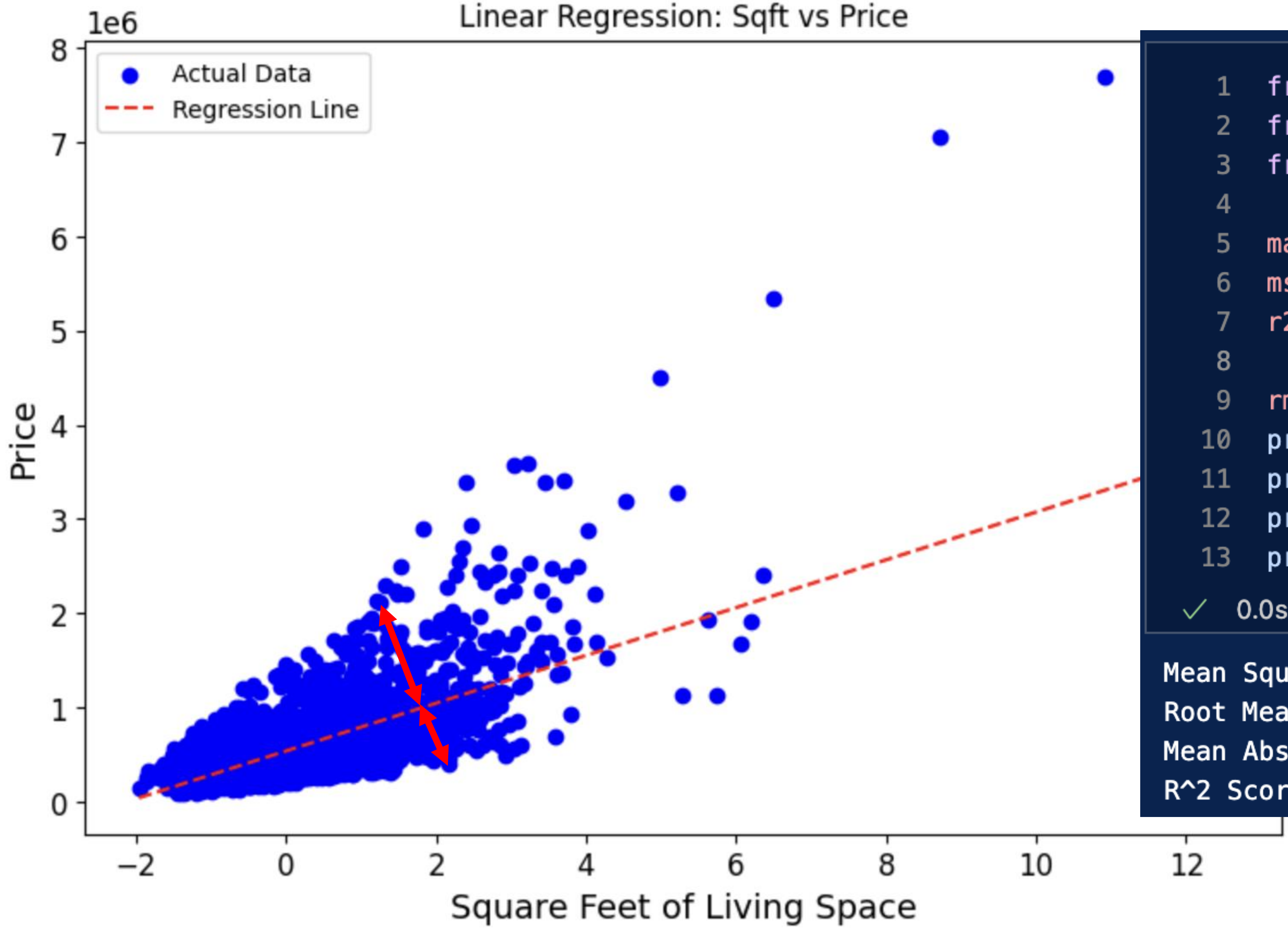
```
Slope (Coefficient): [253563.34]
Intercept: 539396.9331983805
```

```python
y_predictions = model.predict(X_test[['sqft_living']])
print(y_predictions)
```

✓ 0.0s

```
[1386696.44  413907.27  608909.56 ... 1147804.75  819189.8   461685.61]
```

6. Linear Regression: sqft_living vs price




```python
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

mae = mean_absolute_error(y_test, y_predictions)
mse = mean_squared_error(y_test, y_predictions)
r2 = r2_score(y_test, y_predictions)

rmse = np.sqrt(mse)
print("Mean Squared Error (MSE):\t", mse)
print("Root Mean Squared Error (RMSE):\t", rmse)
print("Mean Absolute Error (MAE):\t", mae)
print("R^2 Score (variance explain):\t", r2)
```

✓ 0.0s

```
Mean Squared Error (MSE):        76768545128.07202
Root Mean Squared Error (RMSE):  277071.37190275005
Mean Absolute Error (MAE):       175652.54077940376
R^2 Score (variance explain):    0.49384025432417855
```

Close Form Solution: Normal Equation

- Find the **value of β that minimizes the squared sum of the estimation errors $\boldsymbol{\epsilon}$**
- The issue here is the computational complexity of the explicit solution, especially the complexity of computing with respect to the number of features $(\boldsymbol{X}\text{^}T\ \boldsymbol{X})\text{^}(-1)$ => $O(n\text{^}2.4)\div O(n\text{^}3)$
- A different approach would be to use an **optimisation algorithm** to find the **optimal solution**

$$\widehat{\boldsymbol{\beta}} = \underset{\beta}{\arg\min} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}\|^2$$

$$\Downarrow$$

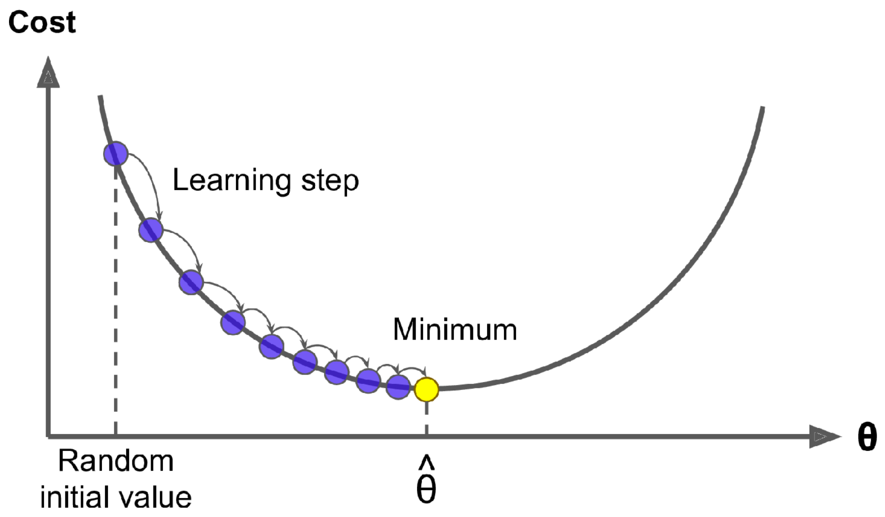
$$\underline{\boldsymbol{X}}^T\,\underline{\boldsymbol{X}}\,\widehat{\boldsymbol{\beta}} = \underline{\boldsymbol{X}}^T\,\underline{\boldsymbol{y}}$$

$$\Downarrow$$

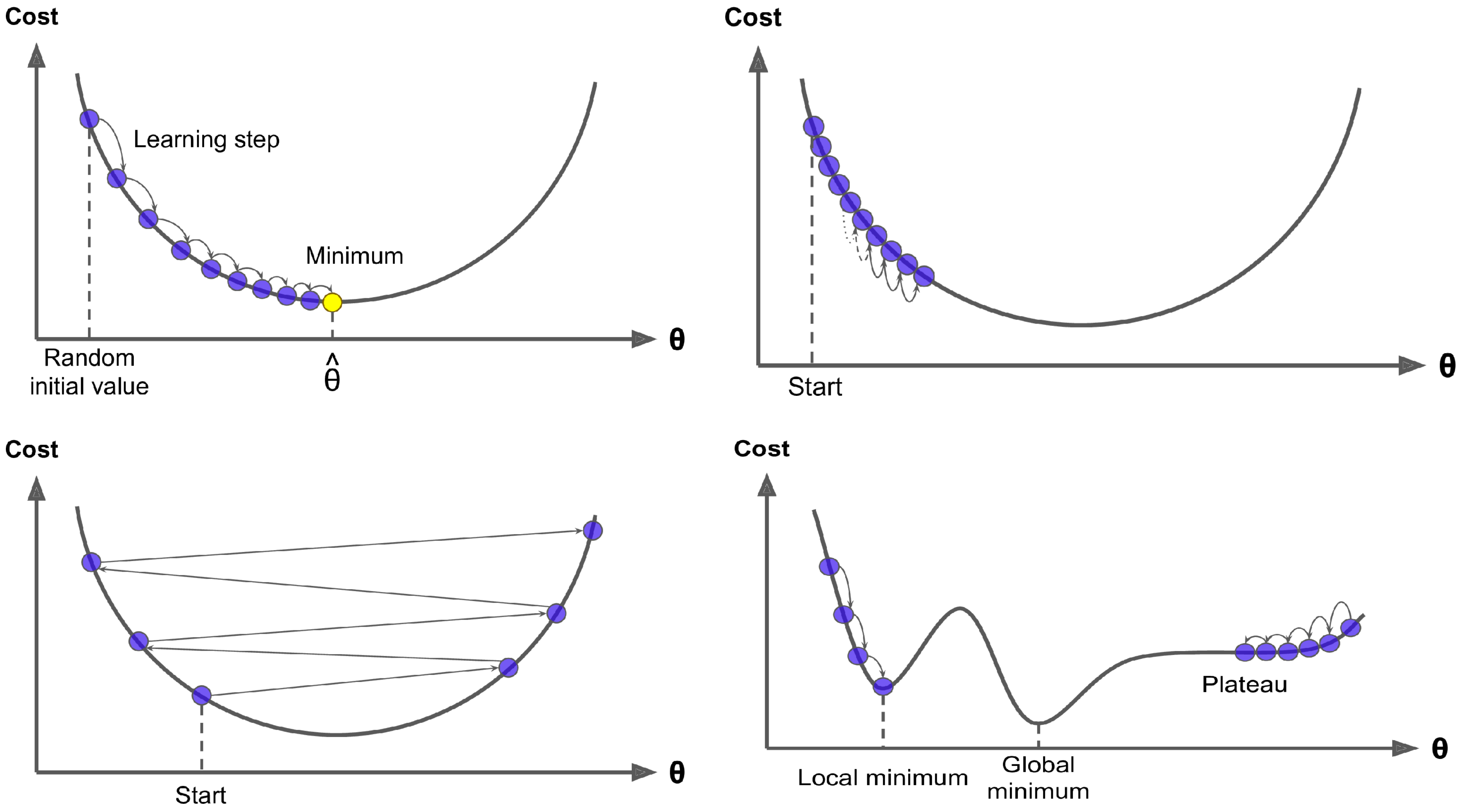
$$\widehat{\boldsymbol{\beta}} = \left(\underline{\boldsymbol{X}}^T\,\underline{\boldsymbol{X}}\right)^{-1}\underline{\boldsymbol{X}}^T\,\underline{\boldsymbol{y}}$$

Gradient Descent

- Tweak (adjust) the weights **β** iteratively in order to **minimize a cost** function.
- Measure the local gradient of the error function with respect to the weights **β**, and tweak **β** in the direction of descending gradient.
- Once the gradient equals zero, you have reached a minimum.



Gradient Descent



```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(
    loss="squared_error", # default cost function (MSE)
    max_iter=2000,   # max numer of epochs. epoch = 1 full iteration over the training set
    penalty=None,
    eta0=1e-3,  # initial learning rate
    tol=1e-3,   # stopping criterion tolerance. stop searching for a minimum
                # (or maximum) once some tolerance is achieved, i.e.
                # once you're close enough.
    random_state=77
)

sgd_reg.fit(X_train, y_train)
print(f"SGD Regressor intercept: {sgd_reg.intercept_})")
print(f"SGD Regressor coefficient: {sgd_reg.coef_}")
```

```
SGD Regressor intercept: [617233.14])
SGD Regressor coefficient: [ -26465.9     3960.11  166890.12    9785.09   -9283.81   47947.42
   22041.67   65475.52   -7549.64 -128317.68 -251401.32  178259.88
 -207806.75 -172014.33  -18649.67  270336.15  428669.8   518157.06
  498916.97  115957.43  -17641.41  -22332.87   -3167.11]
```

7. Hyperparameters (fine tuning) and parameters

- SGDRegressor introduces us to **hyperparameters** (penalty, eta, tol) that can be set and ‘fine tuned’ by humans to optimise the model’s performance.
- There are other **parameters** such as ’intercept’ and ‘coef’ are ‘learnt’ by the model as it is trained – the human does not set these.

```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(
    loss="squared_error", # default cost function (MSE)
    max_iter=2000,   # max numer of epochs. epoch = 1 full iteration over the training set
    penalty=None,
    eta0=1e-3,  # initial learning rate
    tol=1e-3,   # stopping criterion tolerance. stop searching for a minimum
                # (or maximum) once some tolerance is achieved, i.e.
                # once you're close enough.
    random_state=77
)

sgd_reg.fit(X_train, y_train)
print(f"SGD Regressor intercept: {sgd_reg.intercept_})")
print(f"SGD Regressor coefficient: {sgd_reg.coef_}")
```

```python
print("Predictions:", sgd_reg.predict(some_data))
print("Labels:", list(some_labels))
```

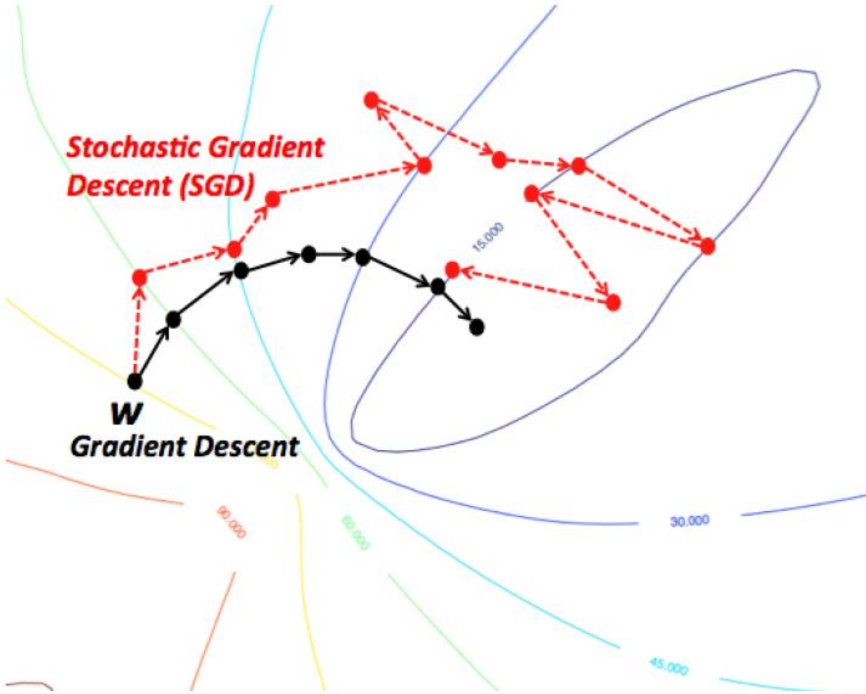
```
Predictions: [ 455016.95  146463.2  1392096.36  210096.8   409172.5   290642.96
  687936.27  230551.52  660043.18  517299.32]
Labels: [379000.0, 173000.0, 1393000.0, 390000.0, 440500.0, 267300.0, 750000.0, 288000.0, 845000.0, 464950.0]
```

```python
y_pred_sgd = sgd_reg.predict(X_train)
sgd_mse = mean_squared_error(y_pred_sgd, y_train)
sgd_rmse = np.sqrt(sgd_mse)
sgd_rmse
```

```
169752.25944316038
```

Stochastic Gradient Descent

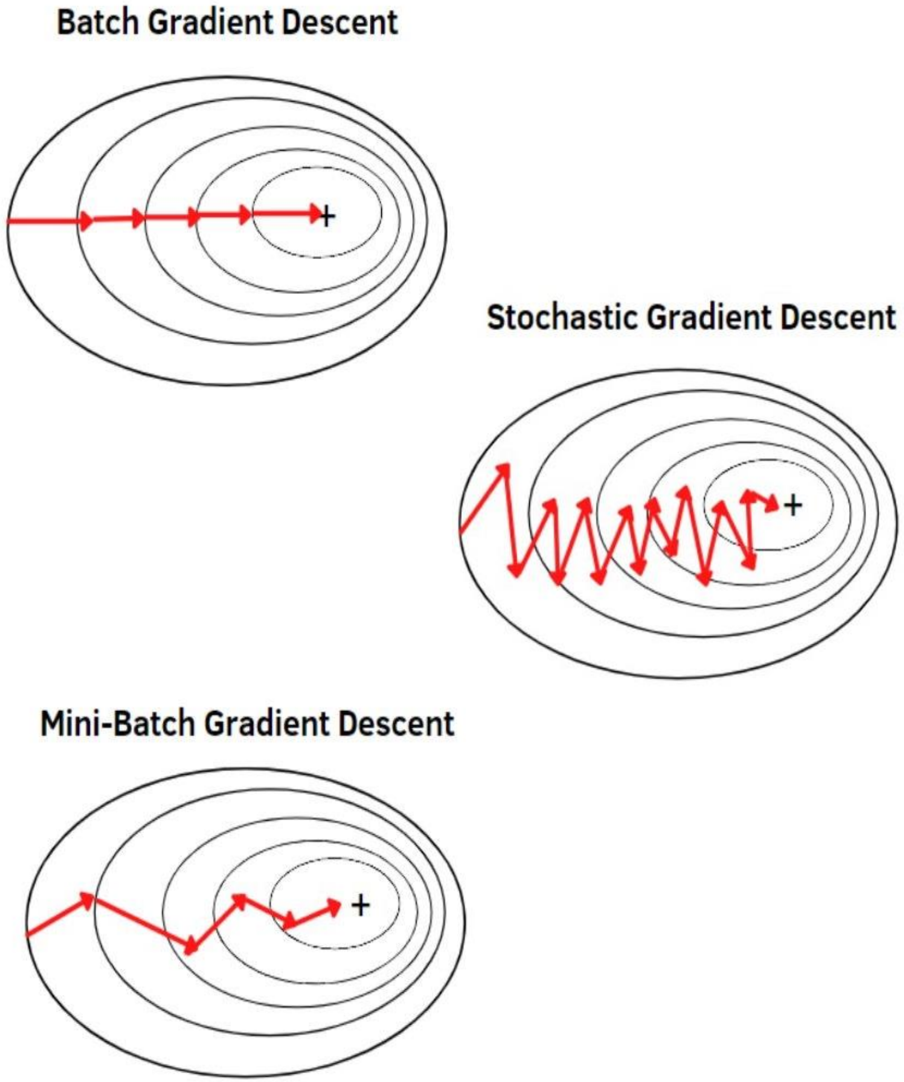
- *Batch Gradient Descent* formula involves calculation over the full training set ***X*** at each Gradient Descent step.
- *Stochastic Gradient Descent (SGD)*: pick a random instance in the training set at every step and computes the gradients based only on that single instance
- SGD: faster algorithm but slower to converge



source: https://wikidocs.net/3413

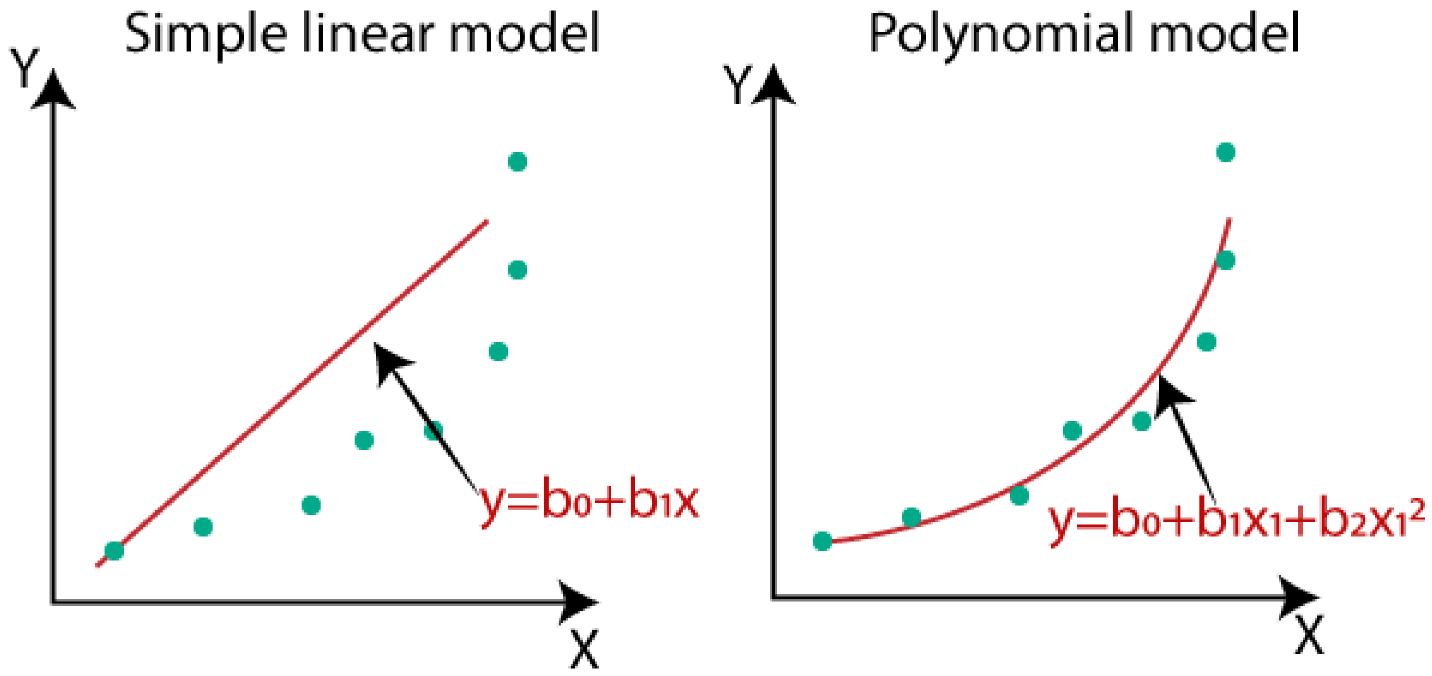
Batch vs Stochastic Gradient Descent

- **Batch gradient descent (BGD)** involves assessing the error for every example in the training dataset but holds off on updating the model until all examples have been processed.
- **Stochastic gradient descent (SGD)** entails both calculating the error and updating the model for each individual example in the training dataset.
- **Mini-Batch** gradient descent divides the training dataset into smaller batches, which are then utilised to compute model error and adjust model coefficients. This method, widely employed in deep learning, strikes a balance between BGD and SGD.



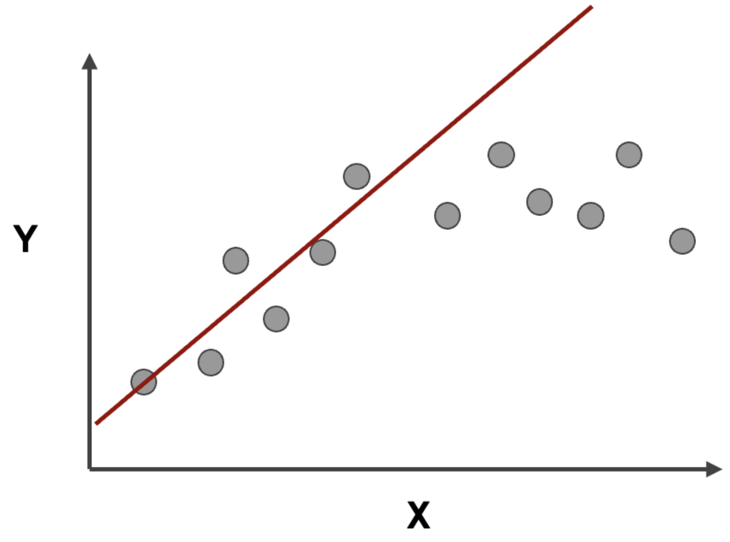
6. Polynomial Regression and Regularisaton

- Data is often more complex than a straight line (or a hyperplane)
- Polynomial regression is an extension of linear regression where we **fit a curve** instead of a straight line by **adding polynomial** terms (x^2, x^3, …) or **'features'** to the model.

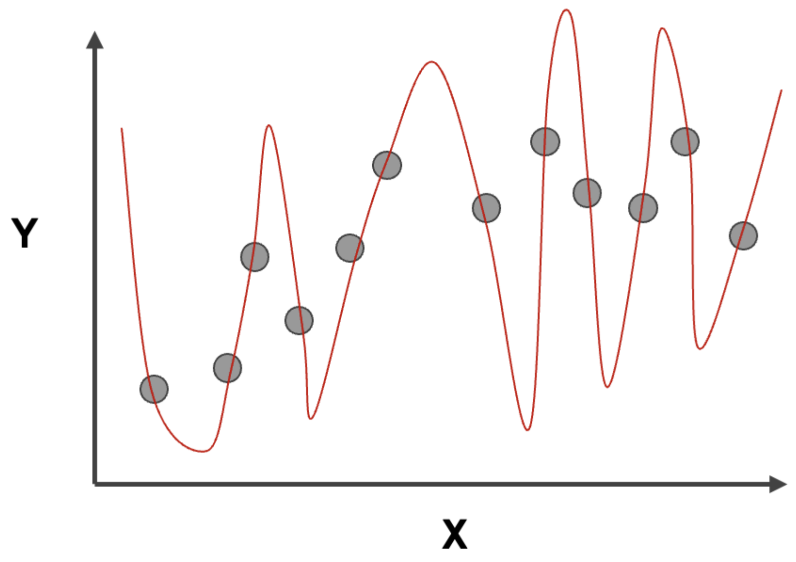


6. Reminder on overfitting and underfitting

- While a **linear model** is prone to **underfit** your data,
- a **polynomial model** may often be prone to **overfitting**
  -> need to use regularisation



Underfitting



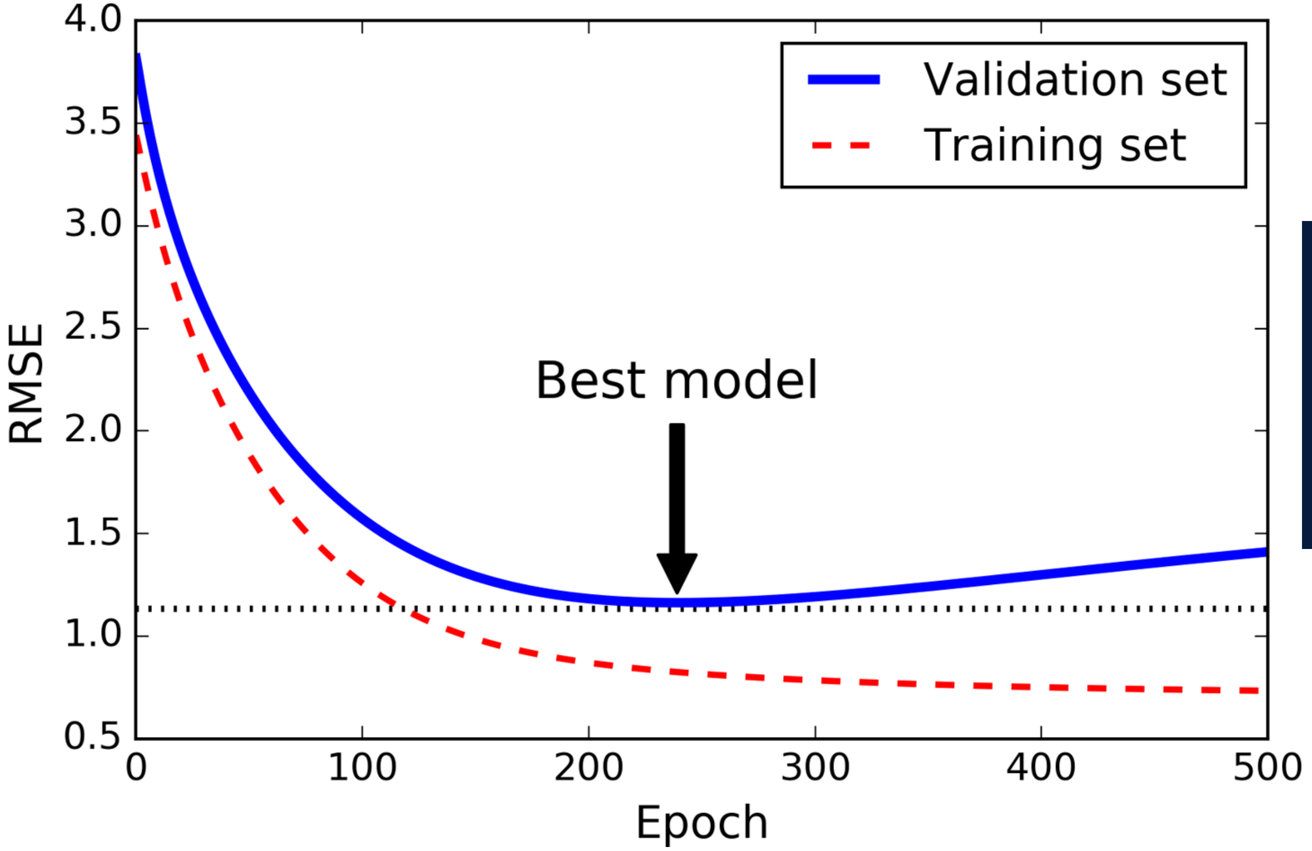
Overfitting

Regularisation

- **What is it?**
  - Technique that constrains our optimization problem to **discourage complex models** and **reduce overfitting**
- **Why do we need it?**
  - Improve generalization of our model on unseen data.
- **Common regularisation techniques:**
  - Dropout layers in Neural Networks (coming up)
  - Early stopping in training models
  - Penalising complexity in regression models

Regularisation: Early Stopping

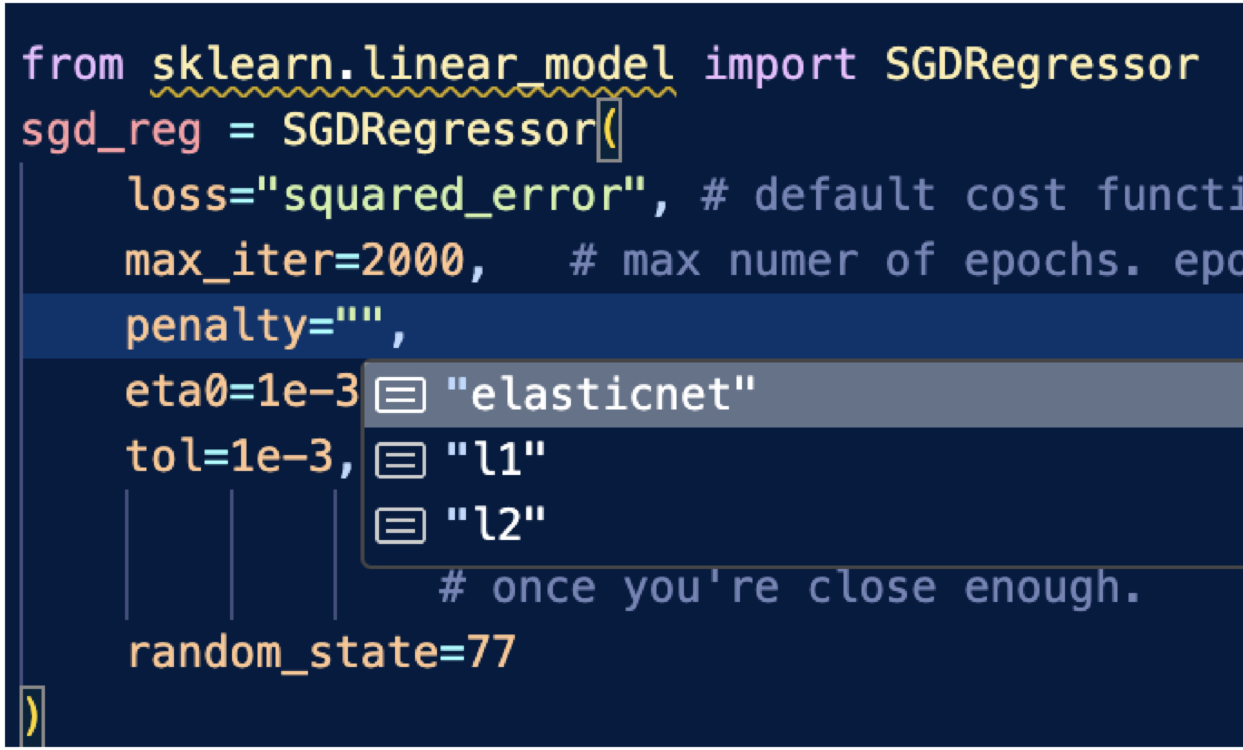
- A way to regularise SGD would be to stop training as soon as the validation error reaches a minimum (before it starts to overfit).




```python
from sklearn.linear_model import SGDRegressor

model = SGDRegressor(early_stopping=True,
                     validation_fraction=0.1,
                     tol=1e-4)
```

Regularisation: Penalising complexity

- SGDRegressor – adjust the ‘penalty’ hyperparameter:
- “l2” for Ridge regression (default, adds squared weights penalty)
- “l1” for Lasso regression (absolute weights)
- “elasticnet” which is a combination of l1 and l2 penalties.
- “None” is no regularisation



Lasso Regression (l1) absolute weights

- These regularization techniques **penalise large model weights**, preventing the model from overfitting.
- Selects features – removes irrelevant features by shrinking these weights **completely to zero.**
- Good for **sparse models**.
- Lasso Regression tends to eliminate the weights of the least important features

```
from sklearn.linear_model import Lasso
ridge_reg = Lasso(alpha=2.5)
cv_res = cross_validate(
    ridge_reg,
    X_train_poly,
    y_train,
    scoring=['neg_root_mean_squared_error', 'r2'],
    cv=k_fold
)
lasso_rmse_scores = -cv_res['test_neg_root_mean_square
display_scores(lasso_rmse_scores)
```

```
Scores: [157093.3  143012.69 140620.71 128170.36 132516.67 1
 138492.77 131973.7  128035.61]
Mean: 136998.46
Standard deviation: 8452.28
```

Ridge Regression (l2) squared weights penalty

- These regularization techniques **penalise large model weights**, preventing the model from overfitting.
- Shrinks weights toward **small but nonzero values**.
- Helps in cases of **multicollinearity**.

```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=25, solver="cholesky")
cv_res = cross_validate(
    ridge_reg,
    X_train_poly,
    y_train,
    scoring=['neg_root_mean_squared_error', 'r2'],
    cv=k_fold
)
cv_res
```

```
{'fit_time': array([0.11, 0.29, 0.29, 0.29, 0.38, 0.3 , 0.3 , 0.29, 0.28
 'score_time': array([0.   , 0.   , 0.   , 0.01, 0.   , 0.   , 0.01, 0.   , 0.
 'test_neg_root_mean_squared_error': array([-143860.6 , -140201.39, -141
        -140817.37, -133473.27, -138920.46, -128476.96, -128185.81]),
 'test_r2': array([0.83, 0.87, 0.84, 0.86, 0.86, 0.85, 0.89, 0.87, 0.86,
```

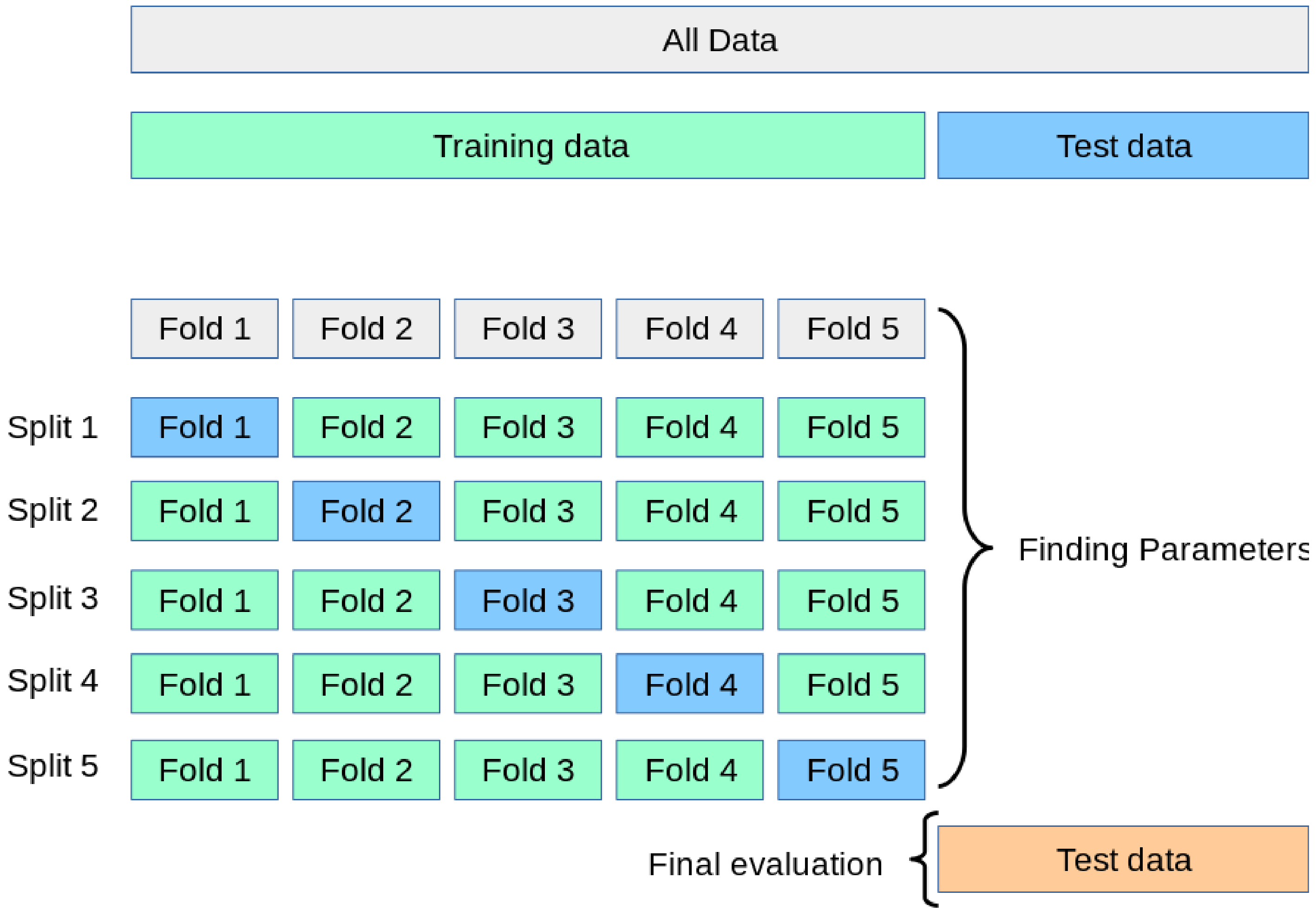
Elastic Net (is combination of Ridge + Lasso)

- Balances feature selection (L1) and generalization (L2).

```python
from sklearn.linear_model import ElasticNet
el_net = ElasticNet(alpha=25, l1_ratio=0.1, max_iter=10000)
cv_res = cross_validate(
    el_net,
    X_train_poly,
    y_train,
    scoring=['neg_root_mean_squared_error', 'r2'],
    cv=k_fold
)
cv_res
```

```
{'fit_time': array([0.57, 0.42, 0.52, 0.38, 0.43, 0.2 , 0.39, 0.36,
 'score_time': array([0.  , 0.  , 0.01, 0.  , 0.  , 0.  , 0.  , 0.
 'test_neg_root_mean_squared_error': array([-273684.95, -290343.6 ,
       -268025.23, -292559.02, -288252.1 , -262258.45, -257086.8 ])
 'test_r2': array([0.38, 0.45, 0.4 , 0.42, 0.43, 0.44, 0.45, 0.42, 0
```

Cross Validation



Cross Validation

```python
from sklearn.model_selection import cross_validate, KFold
n_splits = 10
k_fold = KFold(n_splits=n_splits, shuffle=True, random_state=42)
cv_res = cross_validate(
    lin_reg,
    X_train,
    y_train,
    scoring=['neg_root_mean_squared_error', 'r2'],
    cv=k_fold
)
cv_res
```

```
{'fit_time': array([0.01, 0.01, 0.02, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]),
 'score_time': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]),
 'test_neg_root_mean_squared_error': array([-175290.3 , -183061.83, -167414.87, -156123.95, -169237.03,
        -167950.58, -177667.87, -174713.08, -158683.94, -154060.32]),
 'test_r2': array([0.75, 0.78, 0.78, 0.79, 0.78, 0.78, 0.8 , 0.79, 0.79, 0.78])}
```

Coming up

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