HO CHI MINH UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY FOR HIGH QUALITY TRAINING

Final Report

MACHINE LEARNING

Lecturer: PhD. Tran Nhat Quang

Class: MALE431085E_22_1_01CLC

Subject ID: 221MALE431085E

Members:

Nguyen Minh Tri – 20110422

Phan Thanh Luan – 20110380

Vo Minh Hung – 20110426

Truong Chi Kien – 20110376

Thu Duc city, December 4th, 2022

TABLE OF CONTENTS

INTRODUCTION

We would like to express our gratitude to the professor, PhD. Tran Nhat Quang, who helped us personally throughout the topic-making process in order to successfully complete this topic and this report. We appreciate the teacher's advise from his real-world experience in helping us meet the requirements of the chosen topic, as well as his willingness to always respond to our queries and offer suggestions and corrections. time to assist us in overcoming our flaws and completing it successfully and on time.

We also want to extend our sincere gratitude to the instructors in the High Quality Education Division generally and the Information Technology sector specifically for their committed expertise that has helped us build a foundation. The conditions for learning and performing effectively on the topic have been created by this topic. We also want to express our gratitude to our classmates for sharing expertise and insights that helped us refine our topic. We created the subject and report in a little amount of time, with little expertise and numerous other technical and software project implementation difficulties. Therefore, as it is inevitable that a topic may have flaws, we eagerly await the lecturers' insightful comments in order to further our knowledge and improve for the next time.

We appreciate you very much.

Finally, we would want to wish all of you teachers, ladies and gentlemen, continued health and success in your line of work with developing individuals. Again, we appreciate your kind words.

We sincerely thank you.

COMMITMENT

We would like to assure you that this project was carried out by our team members. We take full responsibility If we commit plagiarism (such as using other people's documents and code without crediting sources, or copying more than 30% of the report).

Full Name	Student ID
Nguyễn Minh Trí	20110422
Phan Thành Luân	20110380
Võ Minh Hưng	20110426
Trương Chí Kiên	20110376

TASK ASSIGNMENT

Student's name	Evaluate contribution	Taskwork
Nguyễn Minh Trí Leader	100%	Making content from Chapter 3 -> Chapter 4, Words Interface. Coding into file ipython notebook.
Võ Minh Hưng	100%	Making content from Chapter 5 -> Chapter 9 Apply difference models to train and test with the given data sets and give the solution for project.
Phan Thành Luân	100%	Making content from Chapter 10 -> Chapter 12. Collect, prepare and process the data set.
Truong Chí Kiên	100%	Making content from Chapter 1 -> Chapter 2. Choose the topic, collect data sets and write description for project.

Note: %: The percentage of each student participating.

No	Goal	Schedule						
1	Understand & decribes the requirements of the project.	o	o					
2	Learn & study technology, language programming to do project	o	o	o				
3	Identify the functions and algorithm to be	o	o	o			o	

	used in the project and report.							
4	Building(Coding) & design of project architecture.	o	o	o				
5	Building & design content of report architecture.			o	o	o		
6	Testing the project.					o	o	o
7	Update and comment the report and project.					o	o	o
Week		2	4	6	8	10	12	14
Note	o-Begin	O – Complete 50%				O – Complete 100%		

Remark of teacher:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Thu Duc city, December 4th, 2022

Signature and full name of teacher

CHAPTER 1: INTRODUCTION OF MARCHINE LEARNING.

1. What is Marchine Learning?

- Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.
- Machine learning is an important component of the growing field of data science. Through the use of statistical methods, algorithms are trained to make classifications or predictions, and to uncover key insights in data mining projects. These insights subsequently drive decision making within applications and businesses, ideally impacting key growth metrics. As big data continues to expand and grow, the market demand for data scientists will increase. They will be required to help identify the most relevant business questions and the data to answer them.
- Examples: Image recognition is a well-known and widespread example of machine learning in the real world. It can identify an object as a digital image, based on the intensity of the pixels in black and white images or colour imagess

2. Type of Marchine Learning

a. With human supervision “or not”

Machine Learning (ML) systems can be classified according to the prior knowledge on the data and type of output it is already known for our training samples.

- Supervised learning us lable data include Classification,e.g.,spam filter, imge classification

Ex: email classification,Image classification,….

- Unspervised learning

- Semisupervised learning

- Reinforcement learning.

b. Learn on the fly “or not”

c. Use Model “or not”

- Instance-based learning: Compares new data to the learned example using a similarity measure

- Model based learning: build a model of the learned exmaples.

CHAPTER 2: END TO END PROJECT.

1. Main step of Marchine learning project

- Look at the picuture.

+ Identify the "End Gold" of the project (game, guess, …)

+ Project analysis, customer opinions,…

- Get the data.

+ Get the data determined in step 1.

+ Some data repositories: UCI Marchine Learning Repository, Amazon's AWS publid datasets or data provided by the customer, …

- Discover and visualize the data to gain insights.

- Prepare the data.

- Train and evaluate models.

- Fine-tune your models.

- Test and analyze your solution.

- Lauch, monitor, and maintain your system.

2. About the end to end project:

- End gold: predict the selling price of apartments and houses

- Part 2: Get the date load data.

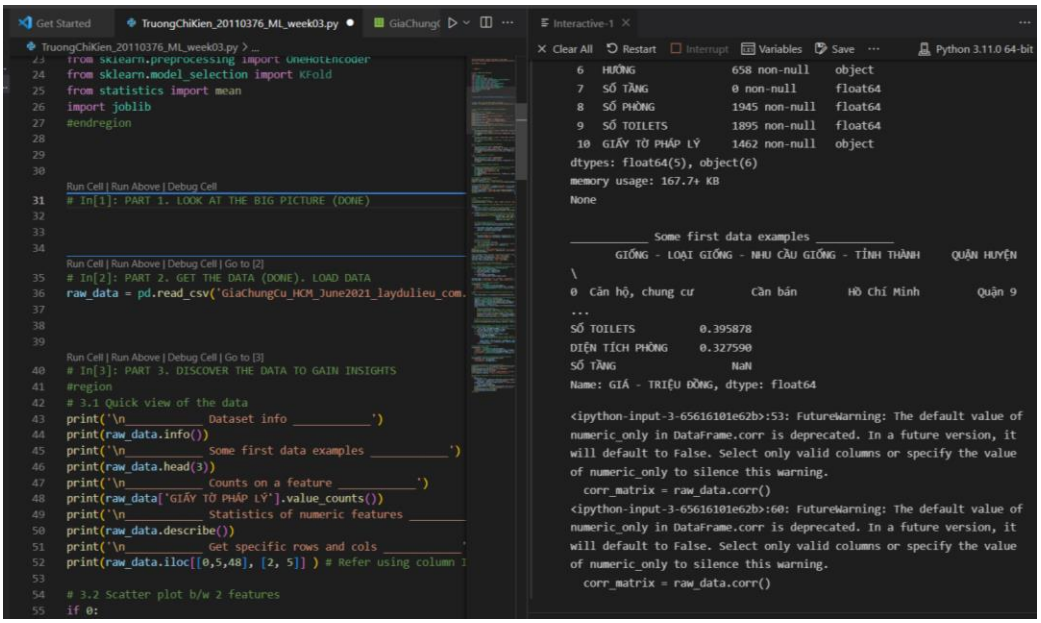
	A	B	C	D	E	F	G	H	I	J	K	L
1	GIỐNG - L	GIỐNG - N	GIỐNG - T	QUẬN HU	GIÁ - TRIỆ	DIỆN TÍCH	HƯỚNG	SỐ TẦNG	SỐ PHÒNG	SỐ TOILET	GIẤY TỜ PHÁP LÝ	
2	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	2650	69			2	2	Đã có sổ	
3	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Tân	3970	74.1			2	2	Đang chờ sổ	
4	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	678	46.5	Tây		1	1	Đang chờ sổ	
5	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Tân	2870	65			2	2		
6	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	3000	70	Đông Bắc		2	2	Đã có sổ	
7	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 7	3200	70			2		Đang chờ sổ	
8	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 2	3800	56.6			2	2		
9	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 11	1570	20	Đông Bắc		1	1	Đã có sổ	
10	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Thủ	3500	89			2	2		
11	Căn hộ, ch	Cần bán	Hồ Chí Min	Huyện Bình	1500	55	Nam		2	2	Giấy tờ khác	
12	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Bình	1750	65			2	2	Đã có sổ	
13	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Bình	1620	66	Tây Bắc		2	1	Đã có sổ	
14	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	1640	50	Bắc		2	1	Đã có sổ	
15	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Tân	1500	60	Nam		2	1	Đang chờ sổ	
16	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 12	1920	70			2	2	Đang chờ sổ	
17	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	2586	68			2	2		
18	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 7	3950	77			2	2	Đã có sổ	
19	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Tân	4700	104.5			3	2	Đang chờ sổ	
20	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 7	2360	57			2	1	Đang chờ sổ	
21	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	910	31			1	1	Đang chờ sổ	
22	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	1400	59	Tây Nam		2	1		
23	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	1293	47			2	1	Đã có sổ	
24	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Tân	3200	88	Bắc		3	2	Đang chờ sổ	
25	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận Bình	2250	83	Bắc		3	2	Đang chờ sổ	
26	Căn hộ, ch	Cần bán	Hồ Chí Min	Quận 9	1256	59	Tây		2	1	Đang chờ sổ	

The data includes features and labels like "Variety-Type", "Price", "Area"

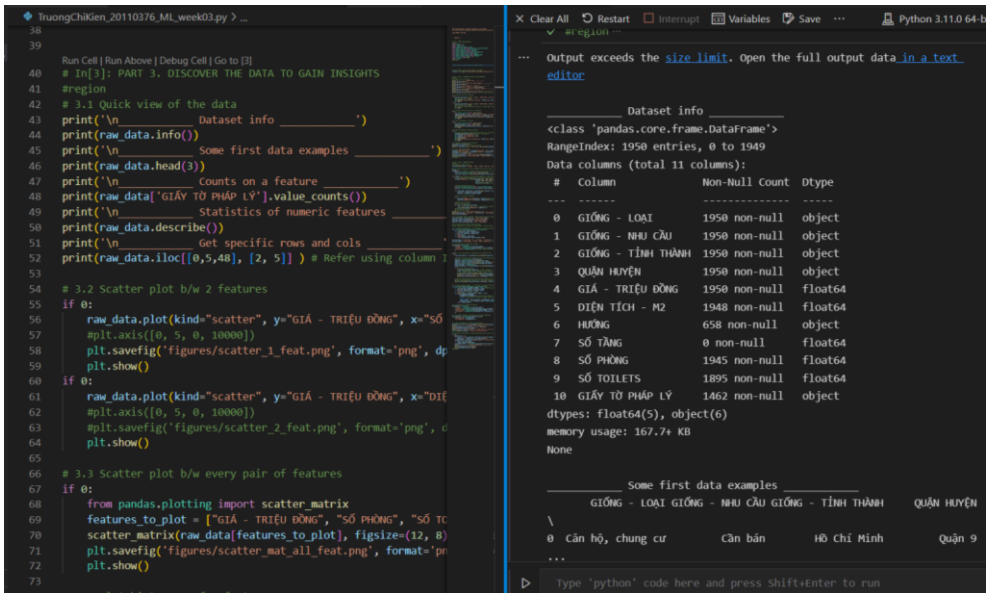
```
Run Cell | Run Above | Debug Cell | Go to [3]
# In[2]: PART 2. GET THE DATA (DONE). LOAD DATA
raw_data = pd.read_csv('GiaChungCu_HCM_June2021_laydulieu_com.
```

➔ Load data: pandas library to help load csv files and read it.

- Part 3



Watch in turn data fame:



➔info algorithm shows us general information about the data for example data has 1950 sample, data type is float or object,…

```
print('\n_____ Some first data examples _____')
print(raw_data.head(3))
```

```
_____ Some first data examples _____
       GIỐNG - LOẠI GIỐNG - NHU CẦU GIỐNG - TỈNH THÀNH     QUẬN HUYỆN
\
0  Căn hộ, chung cư         Cần bán        Hồ Chí Minh         Quận 9
...
SỐ TOILETS          0.395878
DIỆN TÍCH PHÒNG     0.327590
SỐ TẦNG                  NaN
Name: GIÁ - TRIỆU ĐỒNG, dtype: float64
```

➔the algorithm helps to see the first few lines of data

```
print('\n_____ Counts on a feature _____')
print(raw_data['GIẤY TỜ PHÁP LÝ'].value_counts())
```

[4] ✓ 0.4s

```
_____ Counts on a feature _____
Đang chờ sổ     647
Đã có sổ        637
Giấy tờ khác    178
Name: GIẤY TỜ PHÁP LÝ, dtype: int64
```

➔Use “value_count()” to see columns as object, used to check text ex: 647 samples are “Đang chờ sổ”

```
    print('\n_____ Statistics of numeric features _____')
    print(raw_data.describe())
```

✓ 0.1s

```
_____ Statistics of numeric features _____
       GIÁ - TRIỆU ĐỒNG  DIỆN TÍCH - M2  SỐ TẦNG     SỐ PHÒNG   SỐ
TOILETS
count       1950.000000     1948.000000      0.0  1945.000000
1895.000000
mean        3261.196558       75.409320      NaN     2.040617
1.744591
std         5280.514953       70.614754      NaN     0.707758
0.585094
min          100.000000        1.000000      NaN     1.000000
1.000000
25%         1790.000000       56.000000      NaN     2.000000
1.000000
50%         2400.000000       68.000000      NaN     2.000000
2.000000
75%         3450.000000       80.000000      NaN     2.000000
2.000000
max       150000.000000     2400.000000      NaN     8.000000
6.000000
```

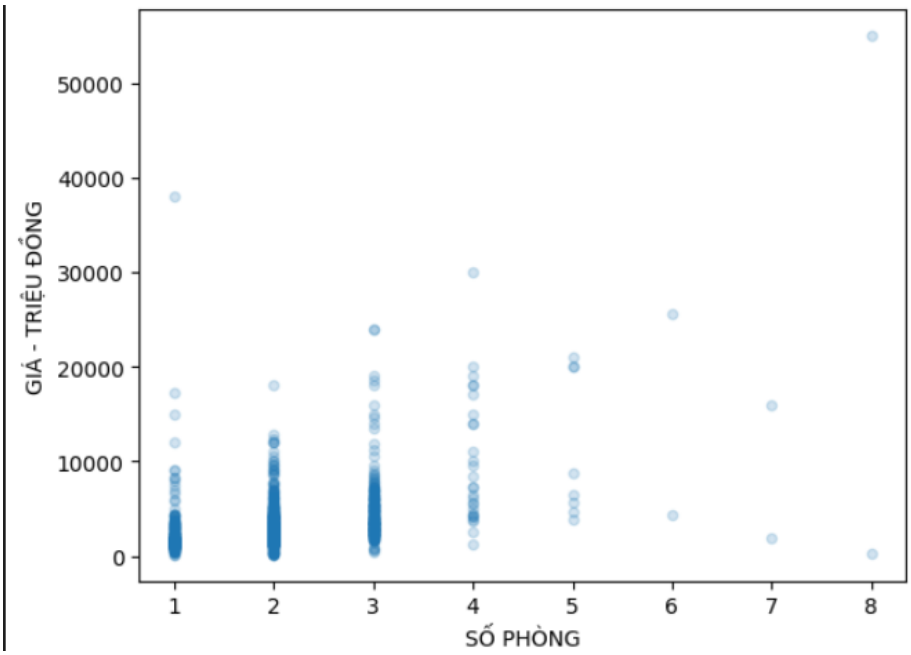
➔function "describe " prints out the statistical information of the numeric column (5 colums of numbers)

```
    print('\n_____ Get specific rows and cols _____')
    print(raw_data.iloc[[0,5,48], [2, 5]] ) # Refer using column ID
```

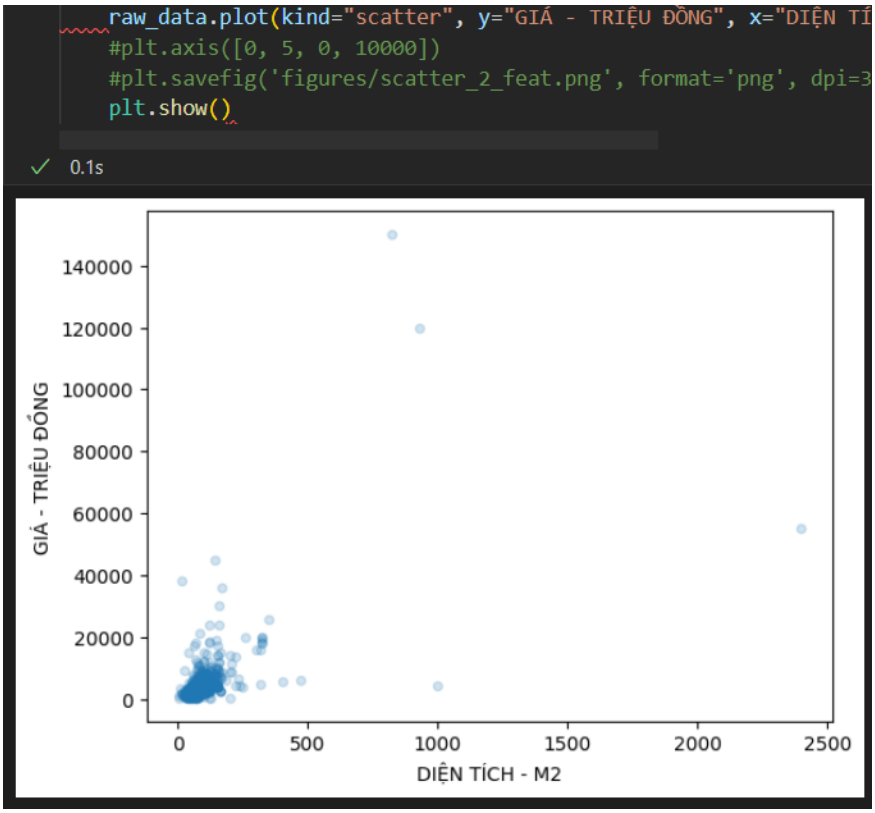
✓ 0.7s

```
_____ Get specific rows and cols _____
   GIỐNG - TỈNH THÀNH  DIỆN TÍCH - M2
0         Hồ Chí Minh           69.0
5         Hồ Chí Minh           70.0
48        Hồ Chí Minh           47.0
```

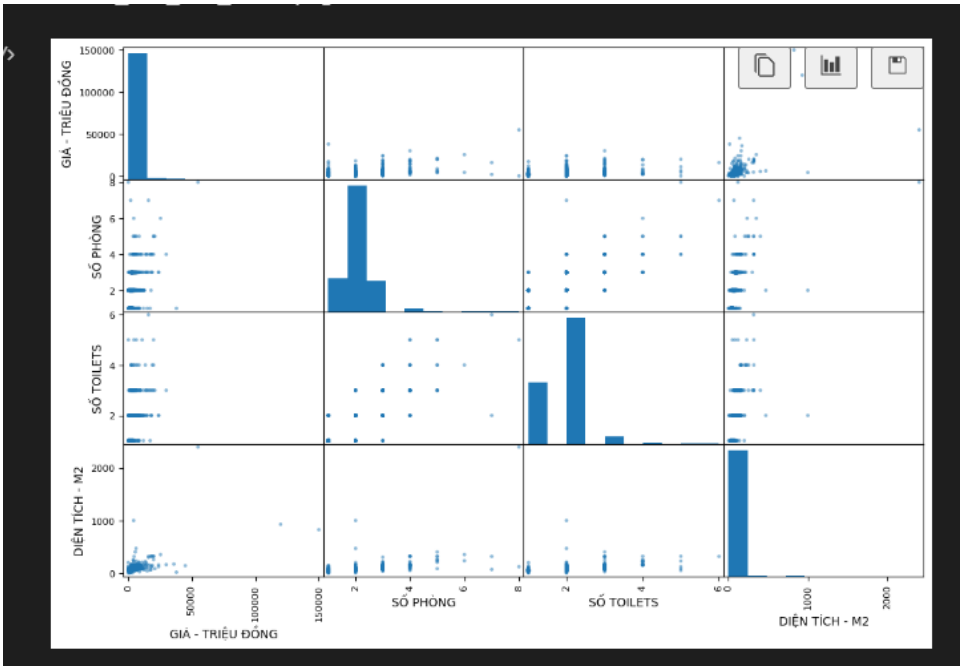
➔"Iloc" specifies the row-array or the array-column to retrieve the element (3 rows 2 columns ==> 6 elements)



➔plot command to draw the figure. Each sample draws a dot. We specify column y as "GIÁ" and column x as "SỐ PHÒNG"



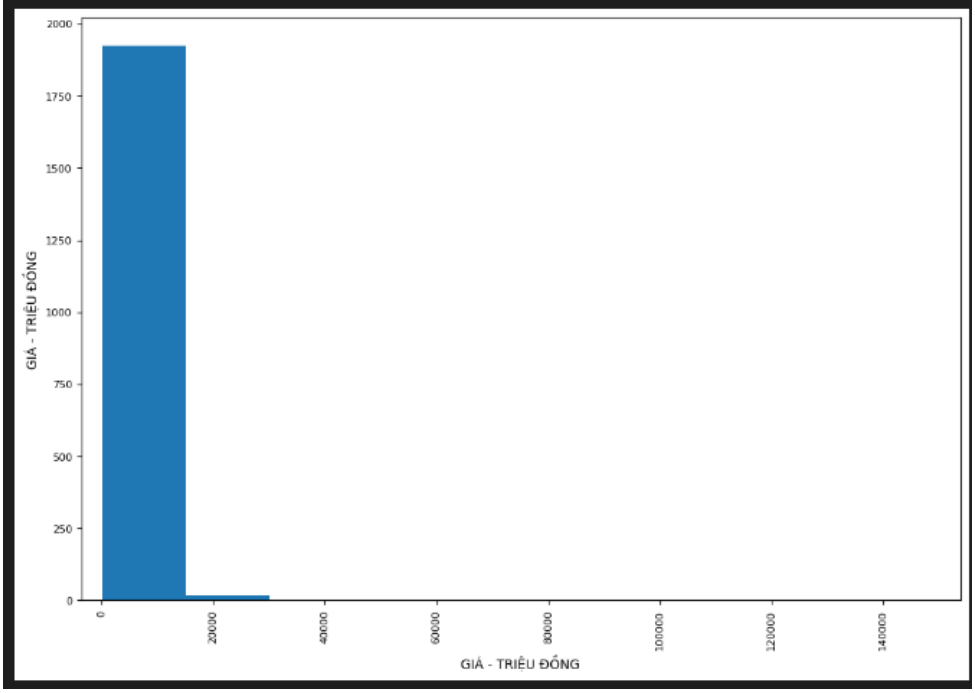
➔Another figure where x is the column "DIỆN TÍCH" and y is "GIÁ"



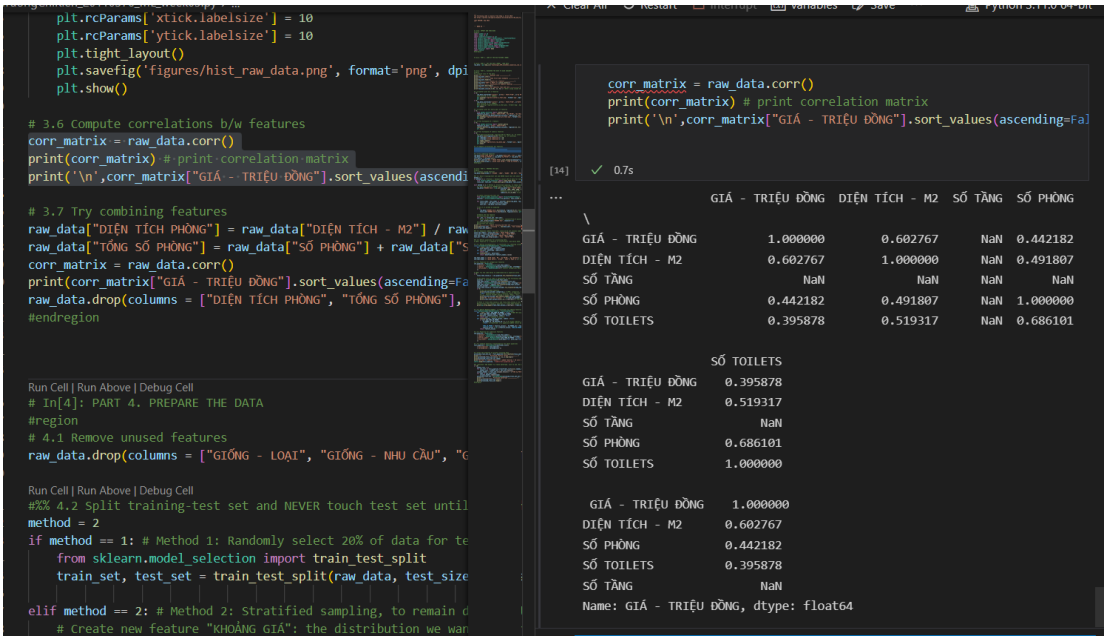
➔In addition, in the pandas library, there is a "scatter_matrix" function that helps to draw all columns. How many columns will have as many pairs

```
from pandas.plotting import scatter_matrix
features_to_plot = ["GIÁ - TRIỆU ĐỒNG"]
scatter_matrix(raw_data[features_to_plot], figsize=(12, 8)) #
plt.show()
```

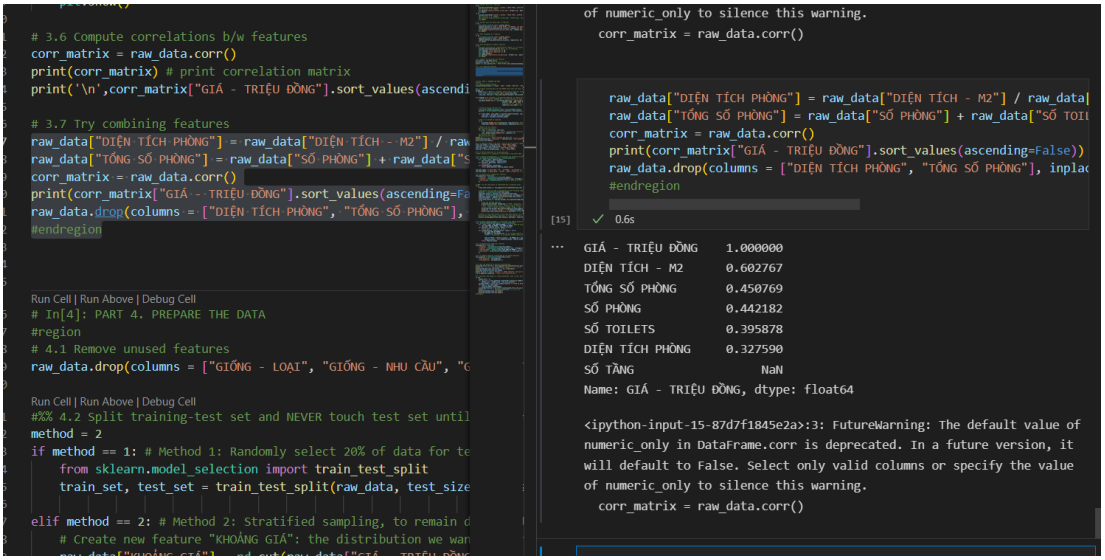
✓ 0.4s



➔the hist command plots all the histograms of the number column



➔Now we apply the "correlations" statistic. corr_matrix will print between all pairs of numeric features in the lesson. e.g. correlation between "GIÁ" and "GIÁ" is definitely equal to 1

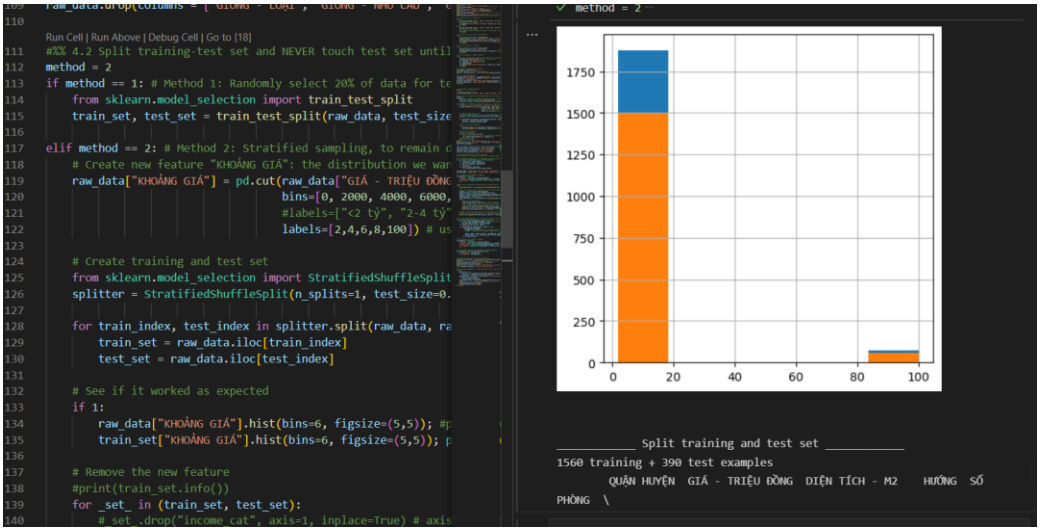


➔create two new features " DIỆN TÍCH PHÒNG" and " TỔNG SỐ PHÒNG" by dividing "DIỆN TÍCH " by " SỐ PHÒNG " and adding the pair "SỐ PHÒNG" with "SỐ TOILETS"

-Part4: prepare the data

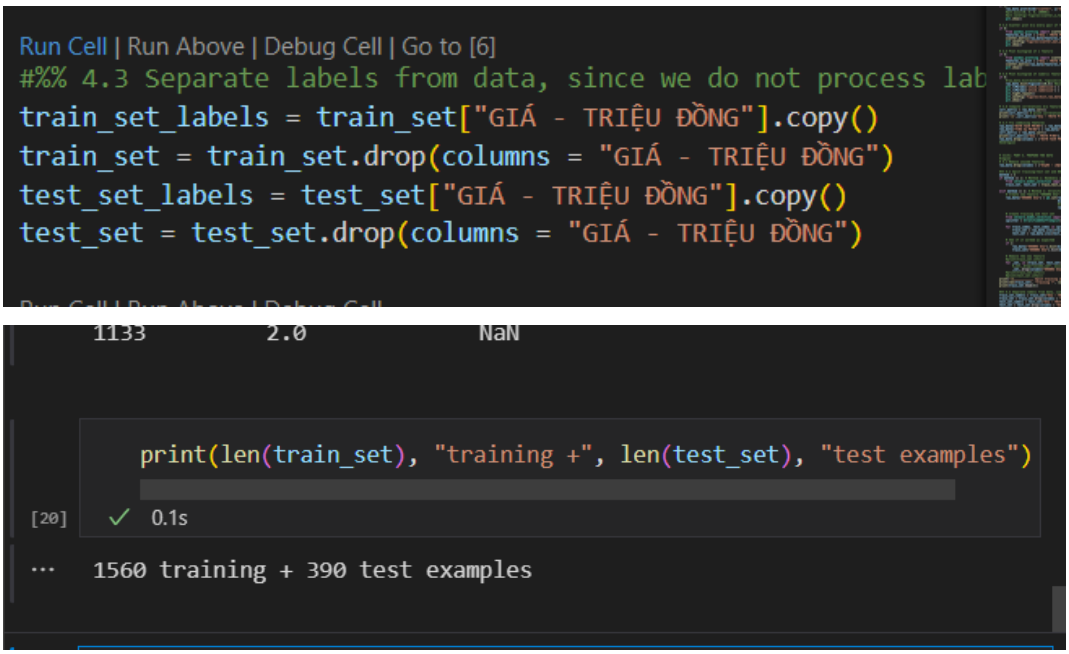
➔delete data columns that are not needed

Before changing the data, we need to do one thing is to separate the "training" and the "test" set. This step gives different results when the data is not split. The reason is that the algorithm can't see it, but when the "test" set is checked, the results are valid



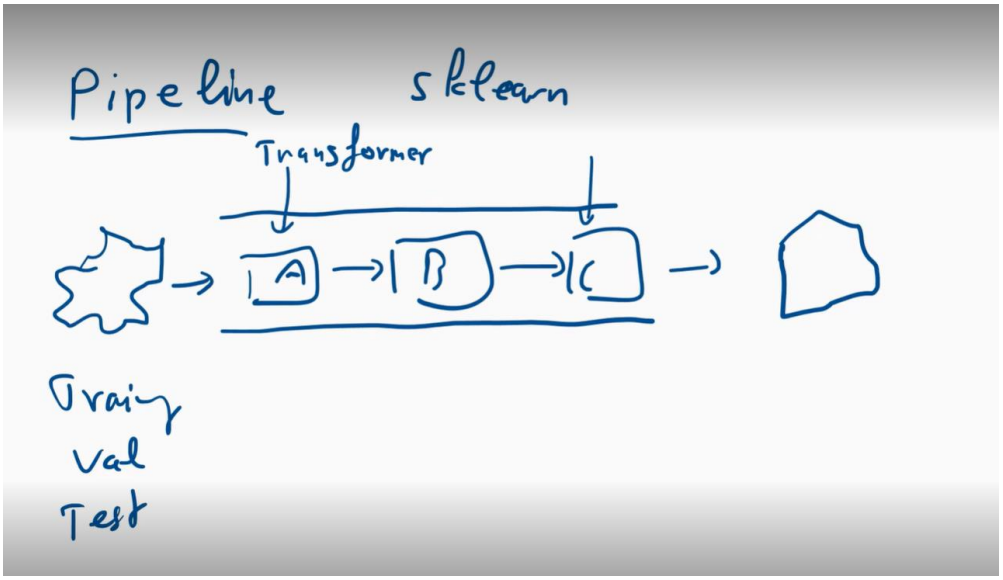
➔Blue is the training set, orange is the test set

After running the above command, the data will become two variables, the set "train_set" and "test_set"



➔This command will separate lables and features

- Next comes the main part of data processing. We are using scikit-learn library, it provides us with a convenient data processing tool called "pipeline"



- Demo

```
Run Cell | Run Above | Debug Cell
#%% 4.4 Define pipelines for processing data.
# INFO: Pipeline is a sequence of transformers (see Geron 2019, page 73). For step-by-step manipulation, see

# 4.4.1 Define ColumnSelector: a transformer for choosing columns
class ColumnSelector(BaseEstimator, TransformerMixin):
    def __init__(self, feature_names):
        self.feature_names = feature_names
    def fit(self, dataframe, labels=None):
        return self
    def transform(self, dataframe):
        return dataframe[self.feature_names].values

num_feat_names = ['DIỆN TÍCH - M2', 'SỐ PHÒNG', 'SỐ TOILETS'] # =list(train_set.select_dtypes(include=[np.nu
cat_feat_names = ['QUẬN HUYỆN', 'HƯỚNG', 'GIẤY TỜ PHÁP LÝ'] # =list(train_set.select_dtypes(exclude=[np.numb
```

➔first create a "transformer" named CollumSelector. Use to separate the data at your discretion. inheriting two classes "BaseEstimator, TransformerMixin"

```
    def __init__(self, feature_names):
        self.feature_names = feature_names
    def fit(self, dataframe, labels=None):
        return self
    def transform(self, dataframe):
        return dataframe[self.feature_names].values
```

➔In a "transformer" there must be at least 3 methods. the first is the constructor of the class, the second is the fit, and the last is the transform function used to transform the data that takes the input as a feature and separates the feature from the numeric and text columns.

```
num_feat_names = ['DIỆN TÍCH - M2', 'SỐ PHÒNG', 'SỐ TOILETS'] # =list(train_set.select_dtypes(include=[np.nu
cat_feat_names = ['QUẬN HUYỆN', 'HƯỚNG', 'GIẤY TỜ PHÁP LÝ'] # =list(train_set.select_dtypes(exclude=[np.numb
```

➔split into two sets of features

```
# 4.4.2 Pipeline for categorical features
cat_pipeline = Pipeline([
    ('selector', ColumnSelector(cat_feat_names)),
    ('imputer', SimpleImputer(missing_values=np.nan, strategy="constant", fill_value = "NO INFO", copy=True)),
    ('cat_encoder', OneHotEncoder()) # convert categorical data into one-hot vectors
    ])
```

➔Next is to define a "pipeline" that first goes through the ColumSelector and then passes in the SimpleImputer (missing_value) and then passes the OneHotEncoder() which turns the letters into numbers.

```
# 4.4.3 Define MyFeatureAdder: a transformer for adding features "TỔNG SỐ PHÒNG",...
class MyFeatureAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_TONG_SO_PHONG = True): # MUST NO *args or **kargs
        self.add_TONG_SO_PHONG = add_TONG_SO_PHONG
    def fit(self, feature_values, labels = None):
        return self  # nothing to do here
    def transform(self, feature_values, labels = None):
        if self.add_TONG_SO_PHONG:
            SO_PHONG_id, SO_TOILETS_id = 1, 2 # column indices in num_feat_names. can't use column names b/c
            # NOTE: a transformer in a pipeline ALWAYS return dataframe.values (ie., NO header and row index)

            TONG_SO_PHONG = feature_values[:, SO_PHONG_id] + feature_values[:, SO_TOILETS_id]
            feature_values = np.c_[feature_values, TONG_SO_PHONG] #concatenate np arrays
        return feature_values
```

➔Next define a new transformer as FeatureAdder, the use of which is to add a column "TỔNG SỐ PHÒNG ".

```
# 4.4.4 Pipeline for numerical features
num_pipeline = Pipeline([
    ('selector', ColumnSelector(num_feat_names)),
    ('imputer', SimpleImputer(missing_values=np.nan, strategy="median", copy=True)), # copy=False: imputatio
    ('attribs_adder', MyFeatureAdder(add_TONG_SO_PHONG = True)),
    ('std_scaler', StandardScaler(with_mean=True, with_std=True, copy=True)) # Scale features to zero mean a
    ])
```

➔4.4.4 defines a pipeline that defines a numeric feature of 4 transformers used to populate column data "TỔNG SỐ PHÒNG".

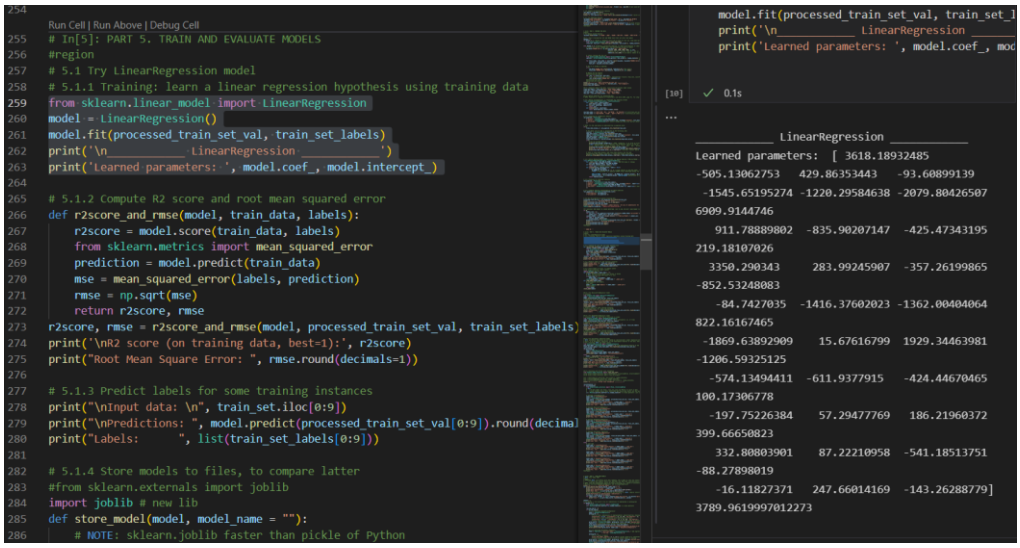
```
# 4.4.5 Combine features transformed by two above pipelines
full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline) ])
```

➔Get a word processing feature and a number processing feature. If we can combine the two, we will have a full pipeline

Get a word processing feature and a number processing feature. If we want to turn the training dataset, we will write fullpipeline.fit_transform(training())f we can combine the two, we will have a full pipeline

- Part5: Train and evalue models.

In this section we will try some algorithms and evaluate it



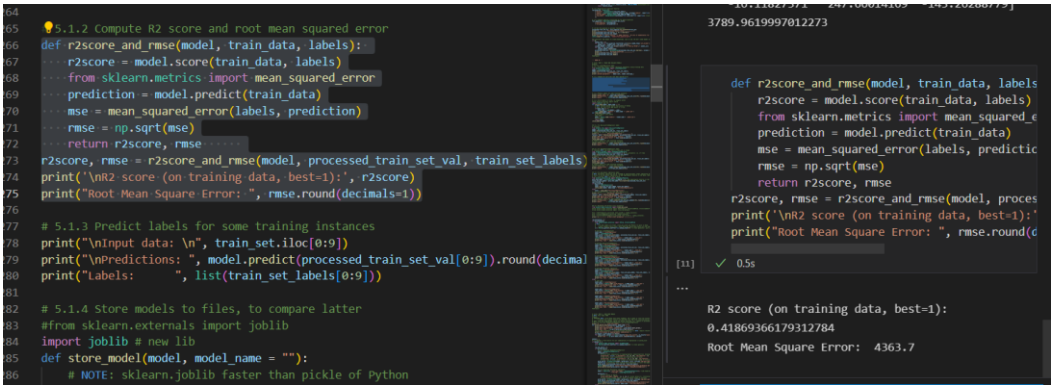
```
Run Cell | Run Above | Debug Cell
# In[5]: PART 5. TRAIN AND EVALUATE MODELS
#region
# 5.1 Try LinearRegression model
# 5.1.1 Training: learn a linear regression hypothesis using training data
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(processed_train_set_val, train_set_labels)
print('\n___________ LinearRegression ___________')
print("Learned parameters: ", model.coef_, model.intercept_)

# 5.1.2 Compute R2 score and root mean squared error
def r2score_and_rmse(model, train_data, labels):
    r2score = model.score(train_data, labels)
    from sklearn.metrics import mean_squared_error
    prediction = model.predict(train_data)
    mse = mean_squared_error(labels, prediction)
    rmse = np.sqrt(mse)
    return r2score, rmse
r2score, rmse = r2score_and_rmse(model, processed_train_set_val, train_set_labels)
print('\nR2 score (on training data, best=1):', r2score)
print("Root Mean Square Error: ", rmse.round(decimals=1))

# 5.1.3 Predict labels for some training instances
print("\nInput data: \n", train_set.iloc[0:9])
print("\nPredictions: ", model.predict(processed_train_set_val[0:9]).round(decima
print("Labels:      ", list(train_set_labels[0:9]))

# 5.1.4 Store models to files, to compare latter
#from sklearn.externals import joblib
import joblib # new lib
def store_model(model, model_name = ""):
    # NOTE: sklearn.joblib faster than pickle of Python
```

```
model.fit(processed_train_set_val, train_set_l
print('\n___________ LinearRegression _____
print('Learned parameters: ', model.coef_, moc

[10] ✓ 0.1s

___________ LinearRegression ___________
Learned parameters:  [ 3618.18932485
-505.13062753   429.86353443   -93.60899139
 -1545.65195274 -1220.29584638 -2079.80426507
6909.9144746
   911.78889802  -835.90207147  -425.47343195
219.18107026
  3350.290343     283.99245907  -357.26199865
-852.53248083
   -84.7427035  -1416.37602023 -1362.00404064
822.16167465
 -1869.63892909    15.67616799  1929.34463981
-1206.59325125
  -574.13494411  -611.9377915   -424.44670465
100.17306778
  -197.75226384    57.29477769   186.21960372
399.66650823
   332.80803901    87.22210958  -541.18513751
-88.27898019
   -16.11827371   247.66014169  -143.26288779]
3789.9619997012273
```

First we'll try some simple "LinearRegression" algorithms for describing data.

To use the algorithm in scikit-learn we just need to import the correct library and then call the models constructor. and then call the .fit function to learn the data (insert features and lables)

- Next, evaluate the model using two performance measures: R2 score and root mean square area



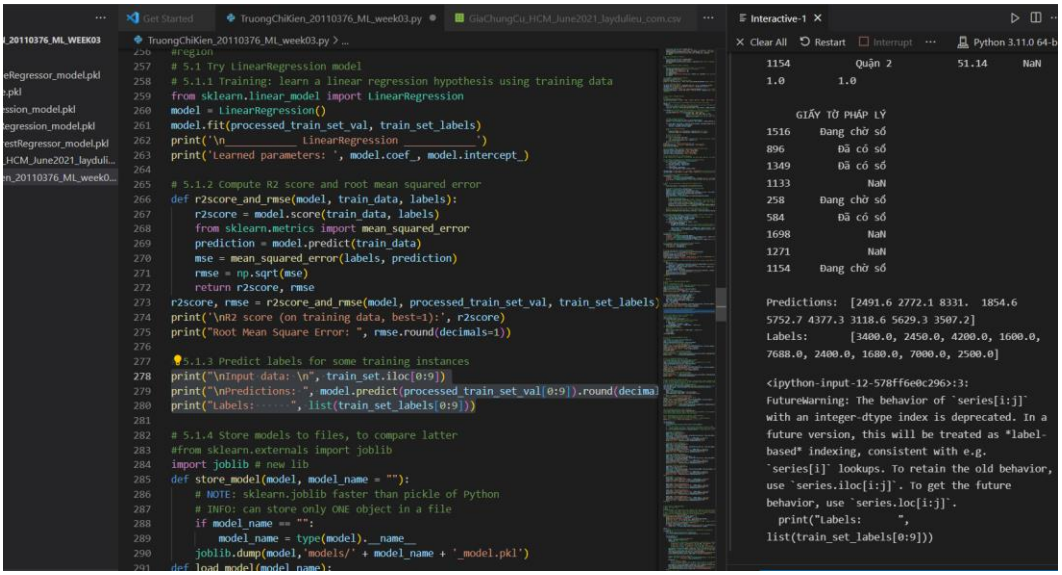
```
# 5.1.2 Compute R2 score and root mean squared error
def r2score_and_rmse(model, train_data, labels):
    r2score = model.score(train_data, labels)
    from sklearn.metrics import mean_squared_error
    prediction = model.predict(train_data)
    mse = mean_squared_error(labels, prediction)
    rmse = np.sqrt(mse)
    return r2score, rmse
r2score, rmse = r2score_and_rmse(model, processed_train_set_val, train_set_labels)
print('\nR2 score (on training data, best=1):', r2score)
print("Root Mean Square Error: ", rmse.round(decimals=1))

# 5.1.3 Predict labels for some training instances
print("\nInput data: \n", train_set.iloc[0:9])
print("\nPredictions: ", model.predict(processed_train_set_val[0:9]).round(decima
print("Labels:      ", list(train_set_labels[0:9]))

# 5.1.4 Store models to files, to compare latter
#from sklearn.externals import joblib
import joblib # new lib
def store_model(model, model_name = ""):
    # NOTE: sklearn.joblib faster than pickle of Python
```

```
3789.9619997012273

    def r2score_and_rmse(model, train_data, labels
        r2score = model.score(train_data, labels)
        from sklearn.metrics import mean_squared_e
        prediction = model.predict(train_data)
        mse = mean_squared_error(labels, predictic
        rmse = np.sqrt(mse)
        return r2score, rmse
    r2score, rmse = r2score_and_rmse(model, proces
    print('\nR2 score (on training data, best=1):'
    print("Root Mean Square Error: ", rmse.round(d

[11] ✓ 0.5s

R2 score (on training data, best=1):
0.41869366179312784
Root Mean Square Error:  4363.7
```

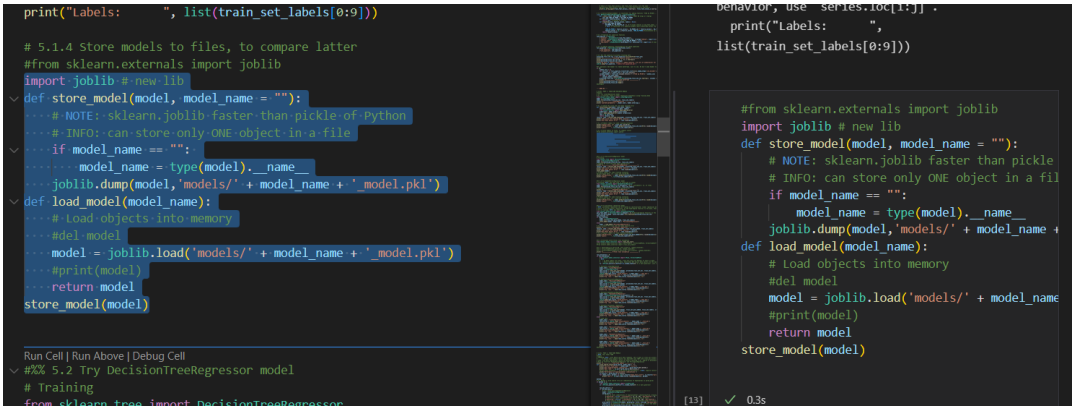
Call the function and then put the data in, then automatically calculate the performance of that model

Which model has R2 score as close to 1 as possible

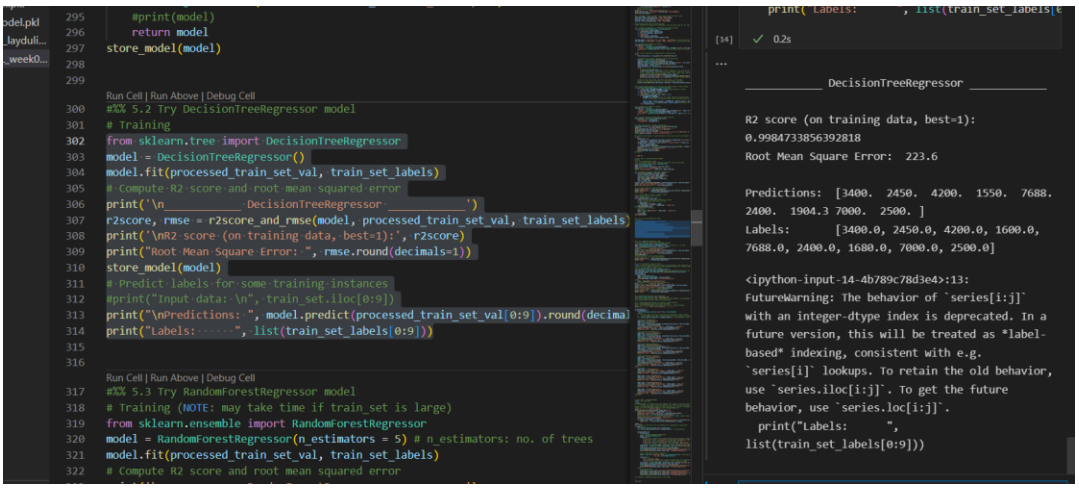
Root mean square area is the error. The smaller the RMSA, the better the model



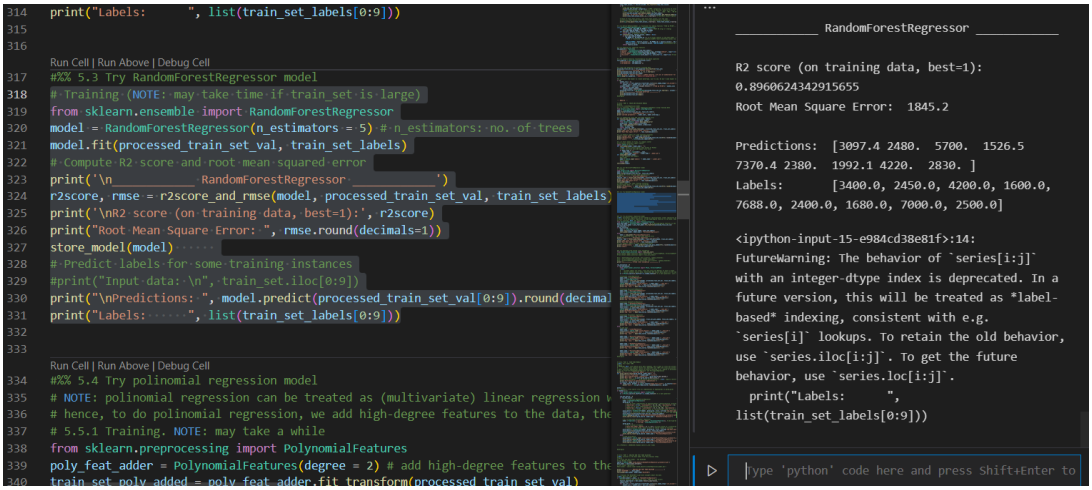
➔Next is the predict function that takes a new feature and automatically returns the result.



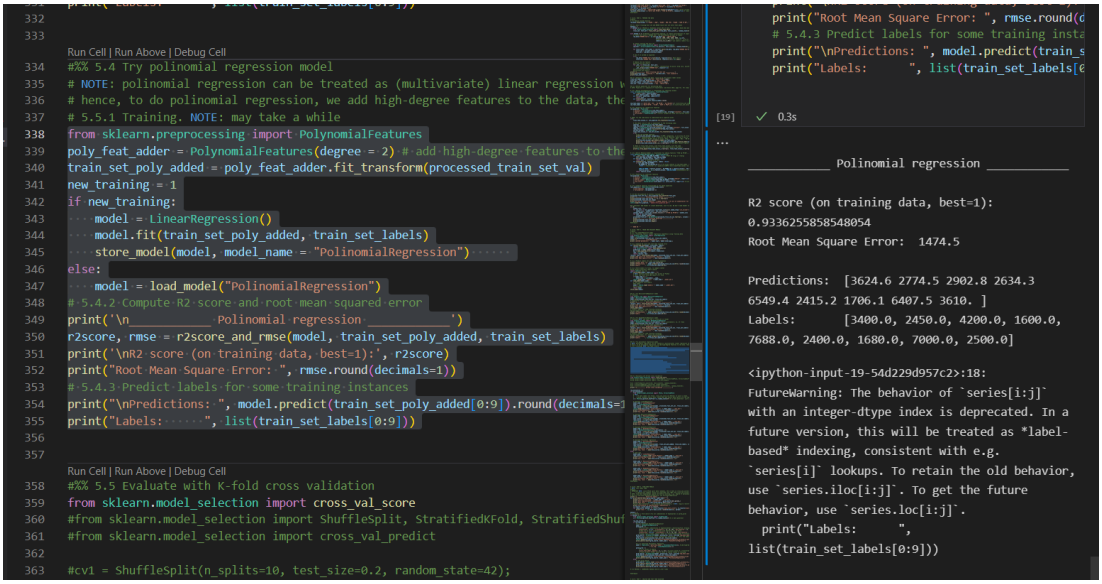
➔Joblib library used to save files.



➔Next is the second model "DecisionTreeRegressor" similar to the first model. This model has R2 score but RMSA is very good. But this model is evaluating on the training set

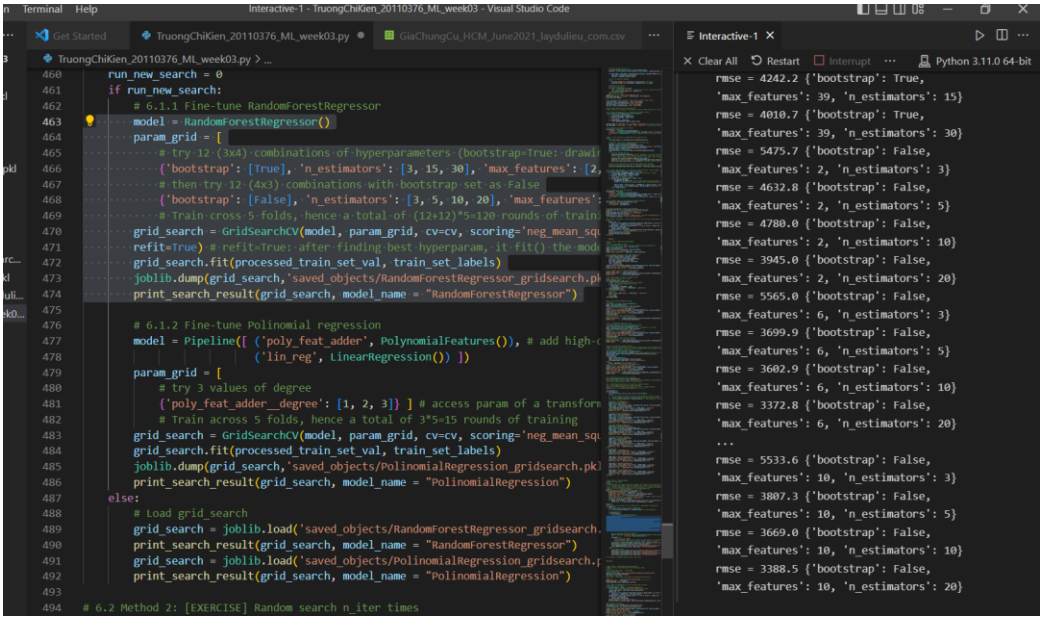


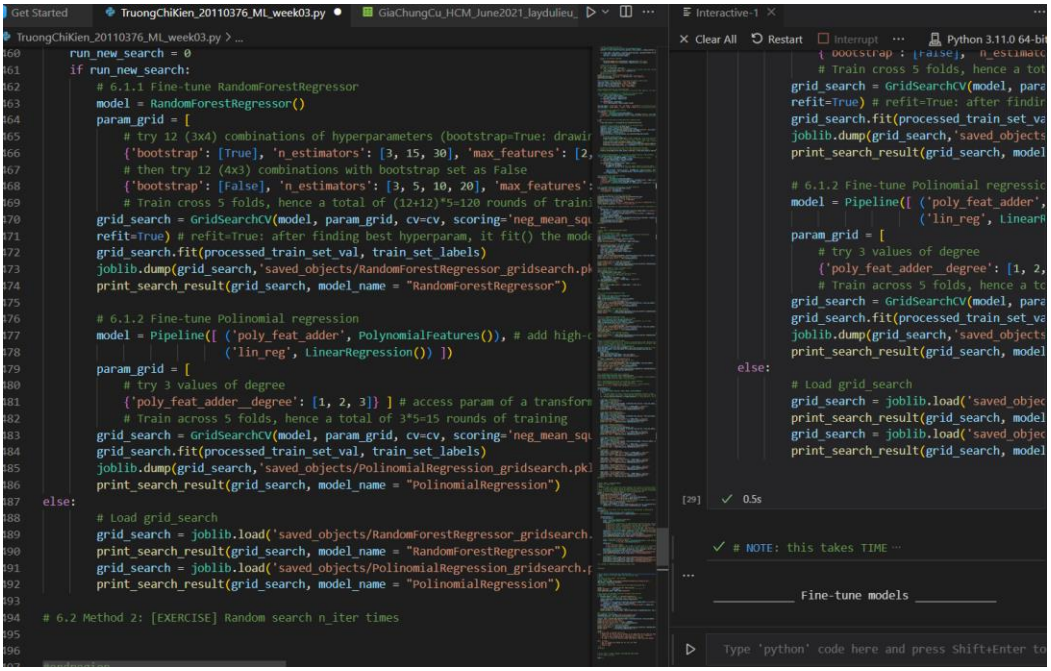
➔Model 3 "Radom forest" One of the best models of marchine learning. This is an algorithm built from many DecisionTree



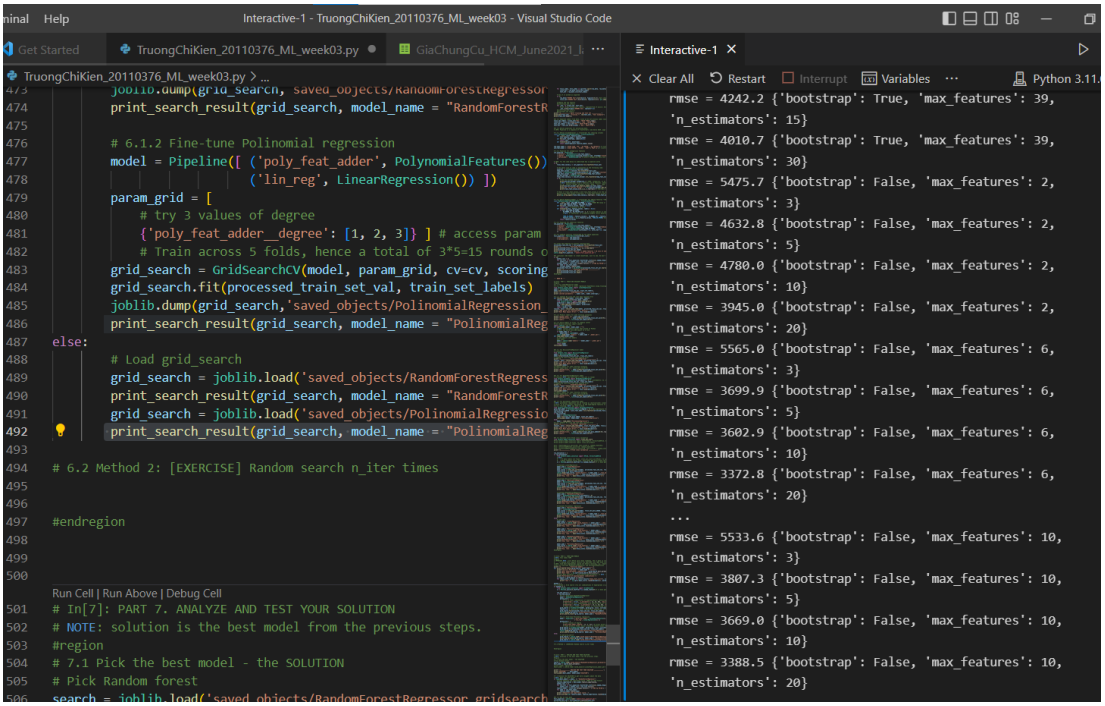
- The last model is PolinomialRegression which uses large-order equations to describe the data. Other uses of the above models. To use step 1 we need to call the class "PolynomialFeatures" and specify the degree we want to select then call the function fit_transform. The results are so good

- Part6:Fine-tune-models



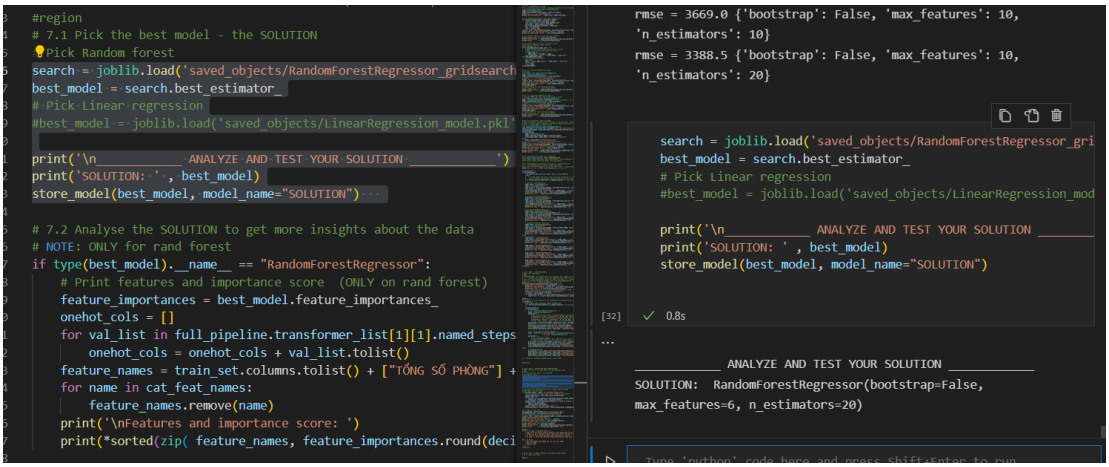


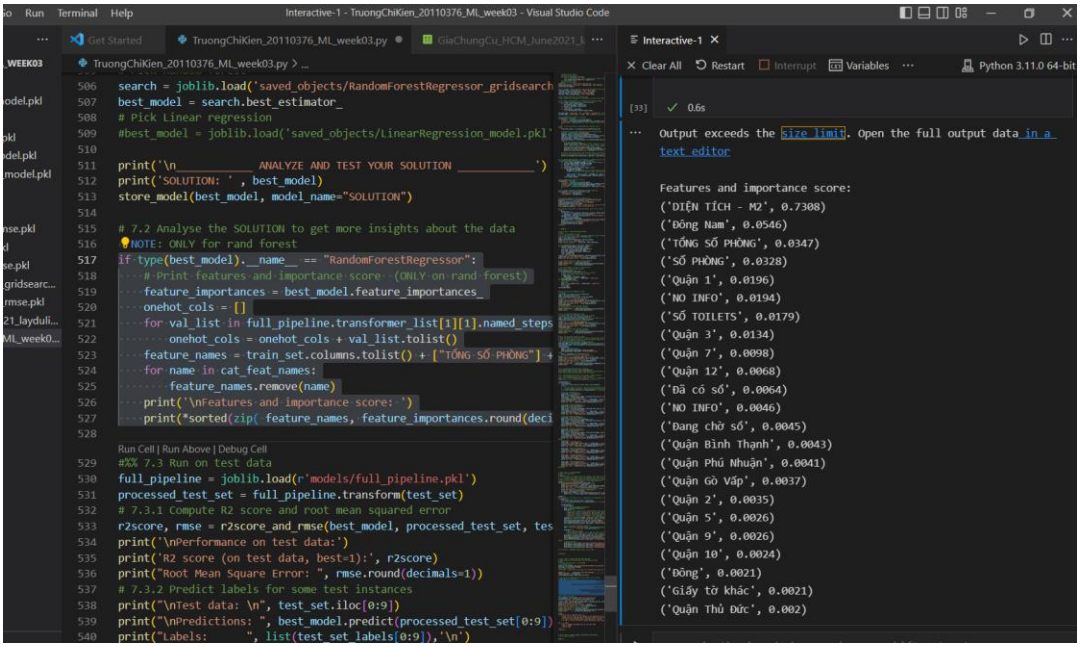
➔Fine-tune-model



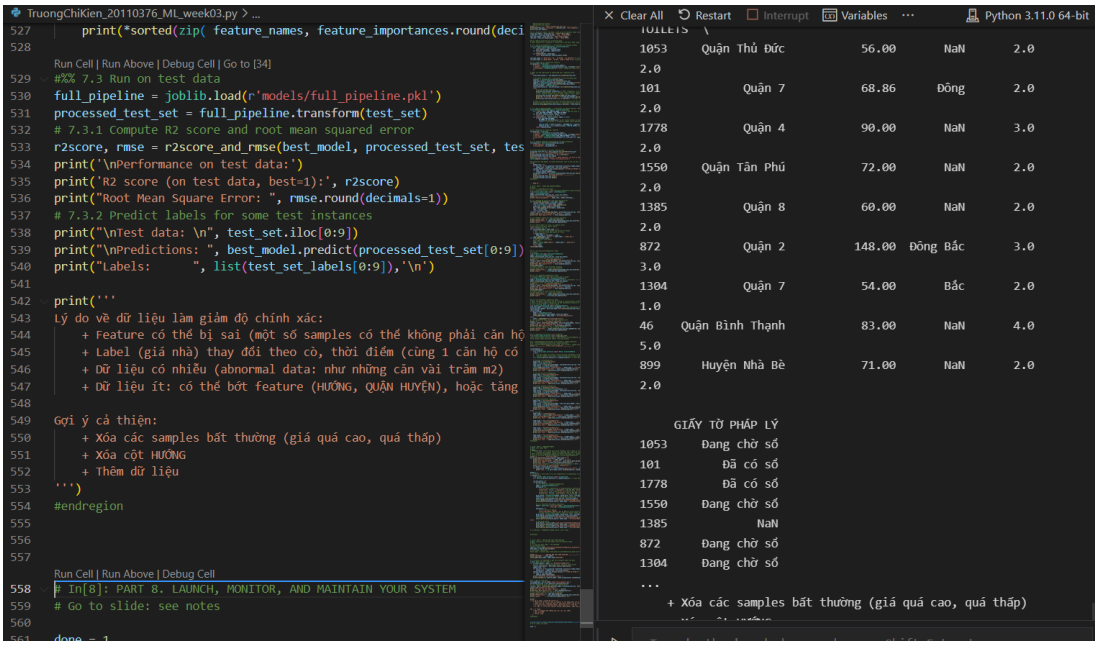
➔predict the error according to the busses and find the best order 1

-Part7: ANALYZE AND TEST YOUR SOLUTION





➔data analysis model, calculate the importance of each feature. Randomforest helps to calculate the importance score. The bigger the importance score, the more influence the feature has on lables.



➔The last step is to run the test set to see how the results are. First reload the pipeline, call the transform function.

CHAPTER 3. CLASSIFICATION

1. Multiclass Classification

1.1. Definition

Multiclass classifiers (also known as multinomial classifiers) may discriminate between more than two classes, in contrast to binary classifiers, which can only distinguish between two classes.

1.2. Multiclass classification strategies

To do multiclass classification using several binary classifiers, you can employ a variety of strategies.

Have 2 type of strategies:

+ One versus All

    Example: Training 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on), is one method for developing a system that can categorize images of the digits into 10 classes (from 0 to 9). When you want to categorize an image, you then obtain the decision scores from each classifier for that image and choose the category whose classifier produces the greatest score. The one-versus-the-rest (OvR) strategy is what it is known as (also called one-versus-all).

+ One versus One

    Example: One alternative is to train a binary classifier to discriminate between each pair of digits: one for 0s and 1s, another for 0s and 2s, another for 1s and 2s, and so on. The one-versus-one (OvO) tactic is known as this. You must train N (N - 1) / 2 classifiers if there are N classes. This entails training 45 binary classifiers for the MNIST problem. To categorize an image, you must submit it to each of the 45 classifiers and determine which class prevails in the most battles. Each classifier only needs to be trained on the portion of the training set for the two classes that it must discriminate, which is the main benefit of OvO.

Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set.

When Scikit-Learn notices that you are attempting to perform a multiclass classification task using a binary classification technique, it automatically executes OvR or OvO, depending on the algorithm.

Example: Scikit-Learn for Support Vector Machine classifier

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

The decision function() method returns 10 scores per instance when you call it (instead of just 1). That's one point for each class:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
```

```
array([[ 2.92492871,  7.02307409,  3.93648529,  0.90117363,  5.96945908,
         9.5       ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

We can see that the class 5 score, which is the highest, is correct.

Node: Use the OneVsOneClassifier or OneVsRestClassifier classes to force Scikit-Learn to use one-versus-one or one-versus-the-rest.

1.3. Error Analysis

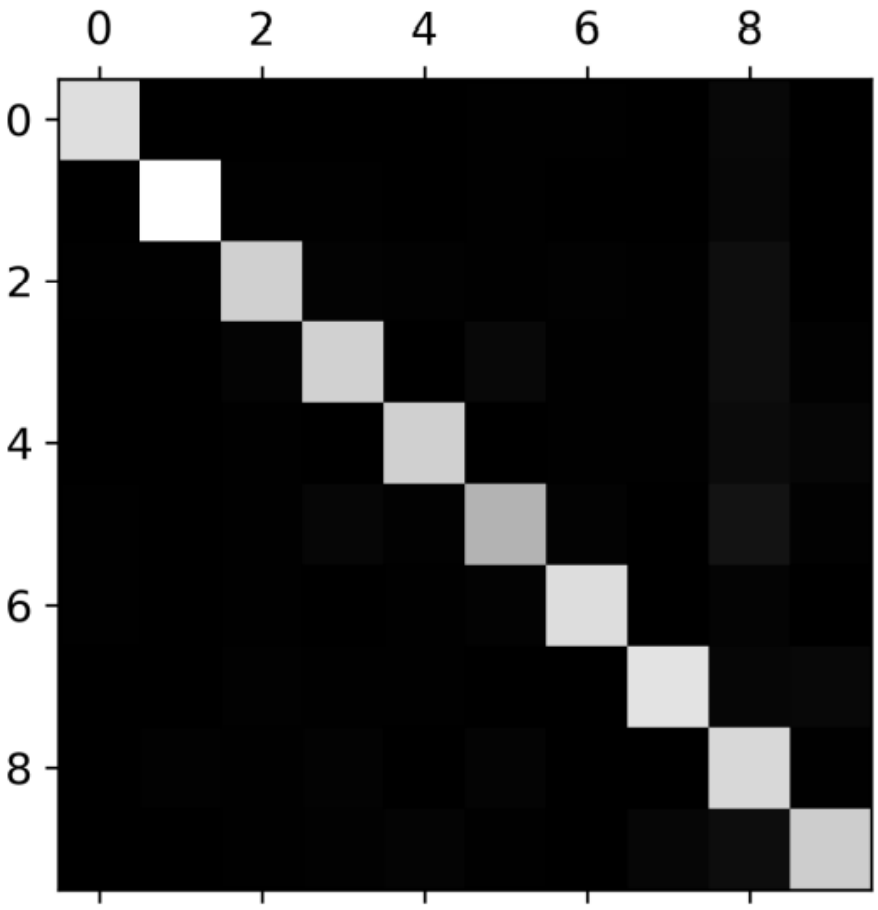
We'll suppose that we've discovered a promising model and are looking for methods to make it better. Analyzing the kinds of faults it commits is one method for doing this.

The cross val predict() function must be used to produce predictions, and then, as before, confusion matrix() must be used.

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

That’s a lot of numbers. It’s often more convenient to look at an image representation of the confusion matrix, using Matplotlib’s matshow() function:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



This confusion matrix looks pretty good, since most images are on the main diagonal, which means that they were classified correctly. The 5s appear a little bit darker than the other numbers, which may indicate that there are fewer images of the 5s in the dataset or that the classifier does not work as well on the 5s as it does on other numbers.

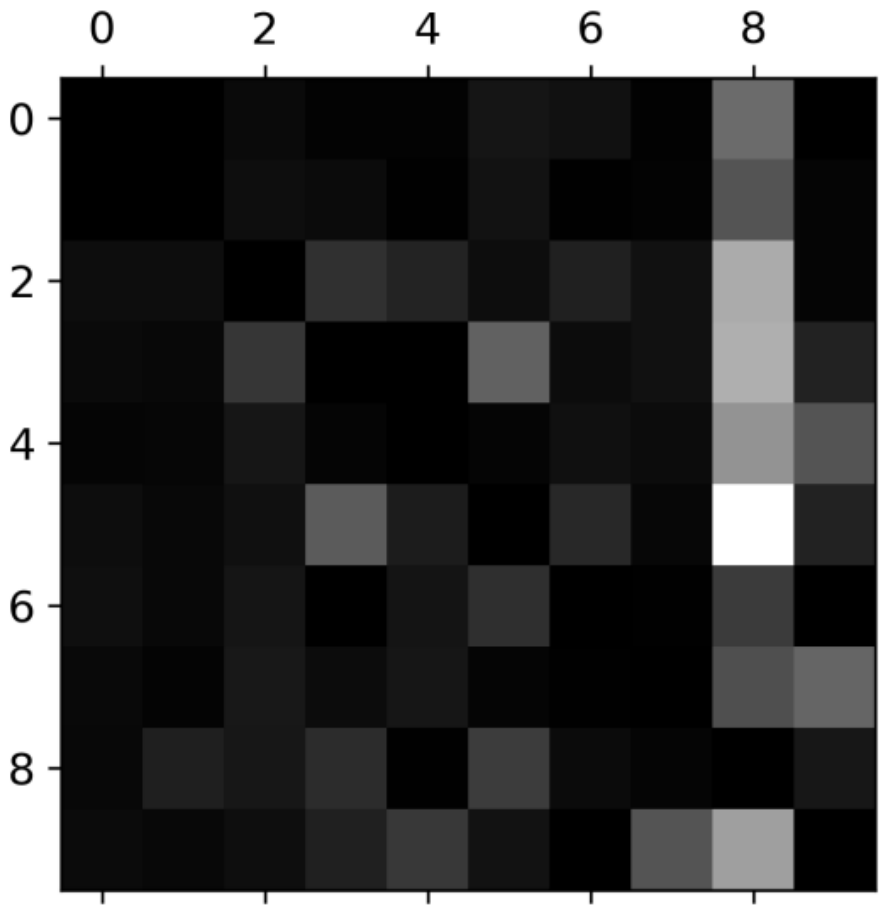
To compare error rates rather than absolute numbers of errors (which would unfairly make numerous classes look poor), divide each value in the confusion matrix by the number of photos in the relevant class:

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Fill the diagonal with zeros to keep only the errors, and plot the result:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```

We can clearly see the kinds of errors the classifier makes:



The column for class 8 is quite bright, which tells us that many images get misclassified as 8s. However, the row for class 8 is not that bad, telling us that actual 8s in general get properly classified as 8s. As we can see, the confusion matrix is not necessarily symmetrical. We can also see that 3s and 5s often get confused (in both directions).

Although it is more complex and time-consuming, analyzing individual errors can also be a useful approach to learn more about what your classifier is doing and why it is failing.

To reduce the false 8s:

+ Gather more training data.

+ Count the number of closed loops.

+ Preprocess the images (e.g., using Scitkit-image, Pillow, or Open CV).

3s and 5s images: SGDClassifier is a linear model.

The main difference between 3s and 5s is the position of the small line that joins the top line to the bottom arc.

Example in the class.

2. Multilabel Classification

A classification system that outputs multiple binary tags is called a multilabel classification system.

For example:

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

For each digit picture, this code generates a y multilabel array with two target labels: the first identifies whether the digit is large (7, 8, or 9), and the second identifies if it is odd. The next few lines build a KNeighborsClassifier instance (not all classifiers enable multilabel classification, though), and we train it using the multiple targets array.

And we can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

And it gets it right! The digit 5 is indeed not large (False) and odd (True).
A multilabel classifier can be evaluated in a variety of ways, and choosing the best statistic actually relies on your project. One method is to calculate the average score after measuring the F score (or any other binary classifier metric) for each unique label. The average F score across all labels is calculated by this code.

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

3. Multioutput Classification

Multioutput Classification is merely a generalization of multilabel categorization in which each label may represent more than one class (i.e., it can have more than two possible values).

For instance:
We start by NumPy's randint() function to add noise to the pixel intensities of the MNIST pictures to create the training and test sets. The original photos will be the target photos:

```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

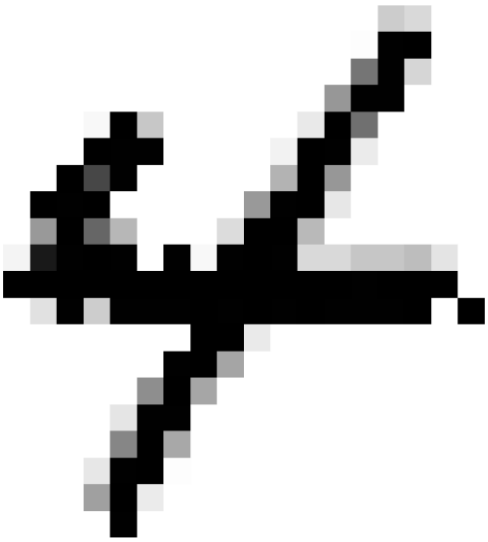
And then we got the result

Noise is in the left and origin (clean) target is in the right.

Ok, we can train the classifier and make it more clean this image:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```

After fixing the noise, we got the clean image like that:



Clean image

CHAPTER 4. TRAINING MODELS

1. Linear Regression

In Chapter 1 we looked at a simple regression model of life satisfaction:

life_satisfaction = θ + θ × GDP_per_capita.

This model is just a linear function of the input feature GDP_per_capita. θ and θ are the model's parameters.

More generally, a linear model generates a prediction by adding the bias term (also known as the intercept term) to the weighted sum of the input features.

Equation 4-1. Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

In this equation:

- ŷ is the predicted value.
- n is the number of features.
- xi is the i-th feature value.
- θ is the j model parameter (including the bias term θ and the feature weights θ , θ , …, θ).

Or we also have the second type of equation:

Equation 4-2. Linear Regression model prediction (vectorized form)

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

In this equation:

- θ is the model's parameter vector, containing the bias term θ and the feature weights θ to θ .
- x is the instance's feature vector, containing x to x , with x always equal to 1.
- θ · x is the dot product of the vectors θ and x, which is of course equal to θ0x0 + θ1x1 + θ2x2 + ⋯ + θnxn.
- h is the hypothesis function, using the model parameters θ.

Setting a model's parameters will make it suit the training set as closely as possible. To achieve this, we must first determine how well (or how poorly) the model matches the training set of data.

The cost function Mean Squared Error (MSE) is the most common performance measure for LR.

The MSE of a Linear Regression hypothesis h on a training set X is calculated

Using this equation:

Equation 4-3. MSE cost function for a Linear Regression model

$$\text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

→ We have to find parameter set θ that minimizes the cost function MSE

1.1. The normal equation:

Take derivatives of the cost function w.r.t. parameters, and equal to zeros.

$$J'(\theta) = \text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}\boldsymbol{\theta} - y^{(i)}\right)^2$$

$$J(\theta_0, \theta_1, \ldots \theta_n) \qquad \left(\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} \ldots + \theta_n x_n^{(i)} - y^{(i)}\right)$$

Equation in the class

There is a closed-form solution, or mathematical equation that gives the answer explicitly, to discover the value of that minimizes the cost function. The Normal Equation is what this is.

Equation 4-4. Normal Equation

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^{\intercal}\mathbf{X})^{-1}\,\mathbf{X}^{\intercal}\,\mathbf{y}$$

In this equation:

ˆ**θ** is the value of θ that minimizes the cost function.

y is the vector of target values containing y^1 to y^m

For instance: We generate some linear-looking data to test this equation.

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

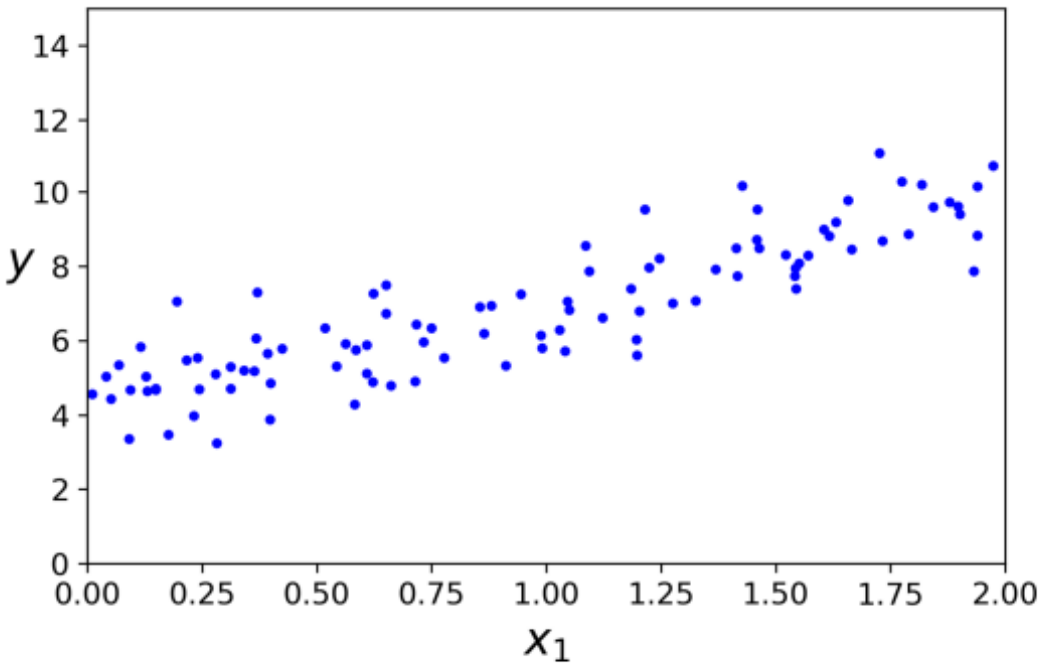


Figure 4-1. Randomly generated linear dataset

Let's now calculate utilizing the Normal Equation. We will compute a matrix's inverse using the inv() function from NumPy's linear algebra module (np.linalg), and we will multiply matrices using the dot() method:

```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

The function that we used to generate the data is y = 4 + 3x + Gaussian noise.

Let’s see what the equation found:

```
>>> theta_best
array([[4.21509616],
       [2.77011339]])
```

We would have hoped for θ = 4 and θ = 3 instead of θ = 4.215 and θ = 2.770.

Now we can make predictions using **θ**ˆ

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

And then we plot this model's predictions:

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

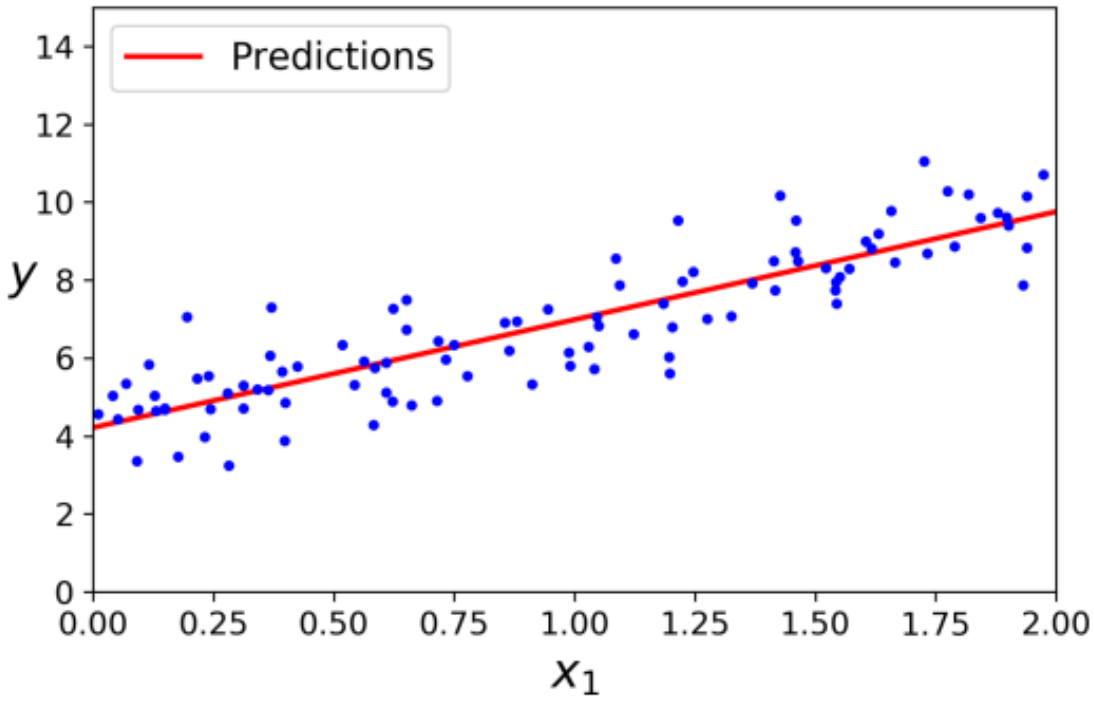


Figure 4-2. Linear Regression model predictions

Computational Complexity:

The Normal Equation computes the inverse of X^T*X, which is an (n + 1) × (n + 1) matrix (where n is the number of features). The computational complexity of inverting such a matrix is typically about O(n) to O(n), depending on the implementation.

1.2. Gradient Descent

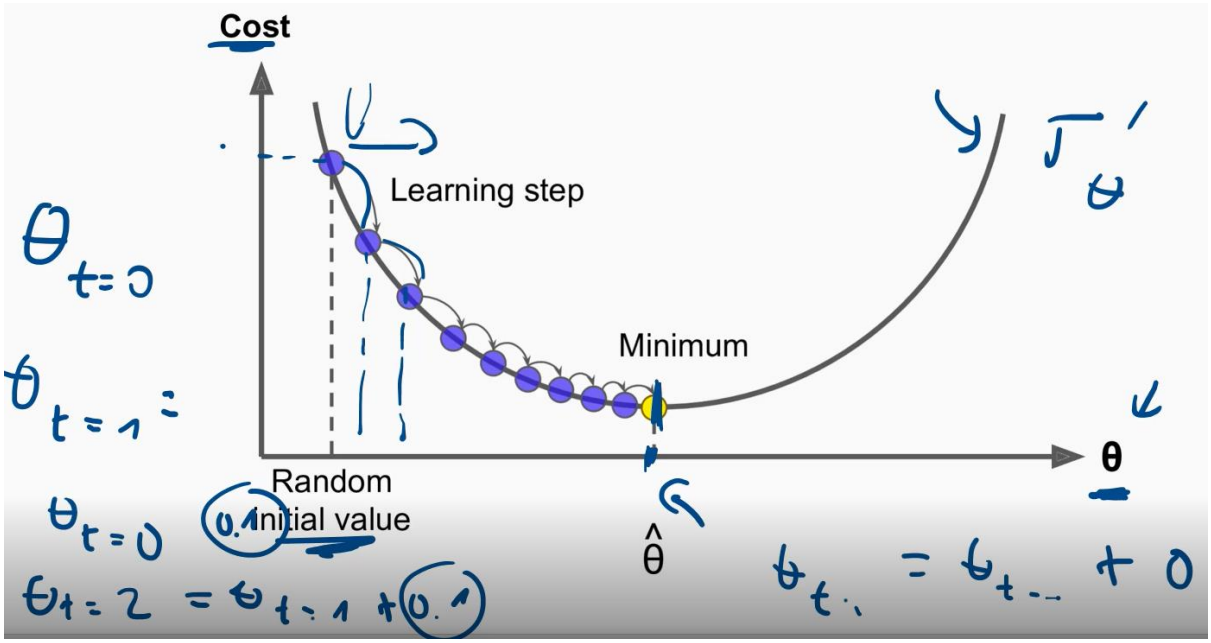
Definition

Gradient Descent is a general-purpose optimization process that may locate the best answers to a variety of issues.

Gradient Descent's main principle is the iterative adjustment of parameters to minimize a cost function.

Idea of Gradient Descent

Gradient Descent calculates the local gradient of the error function with respect to the parameter vector and moves in the falling gradient direction. We have arrived at a minimum when the gradient is zero!

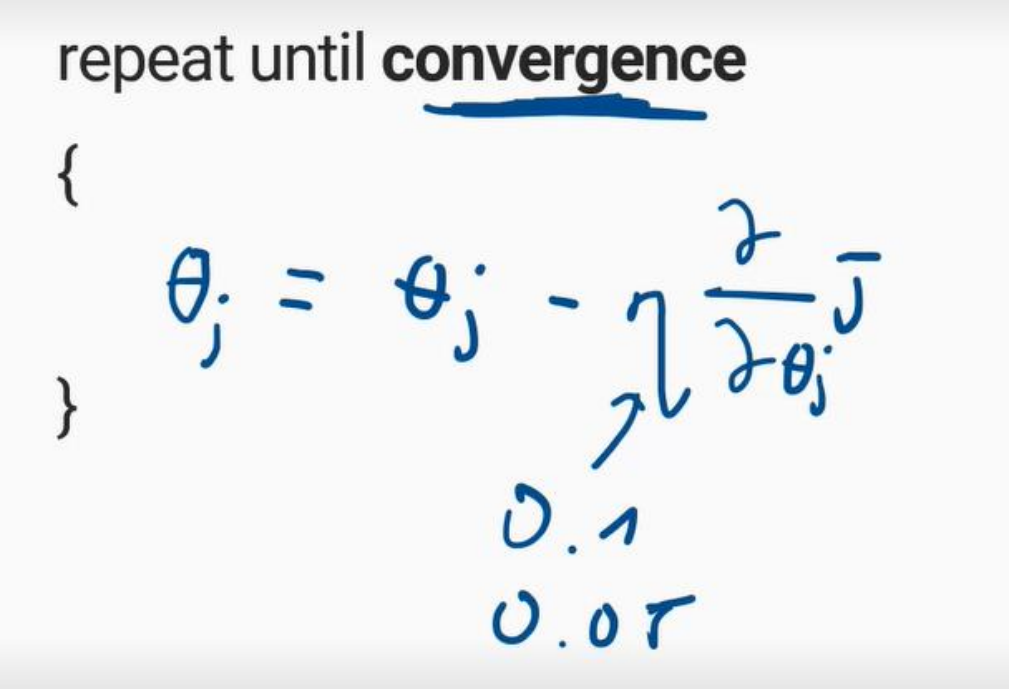


Slide in the class

The learning step size is related to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum in this illustration of gradient descent. The model parameters are randomly initialized and are adjusted periodically to minimize the cost function.

Gradient descent algorithm

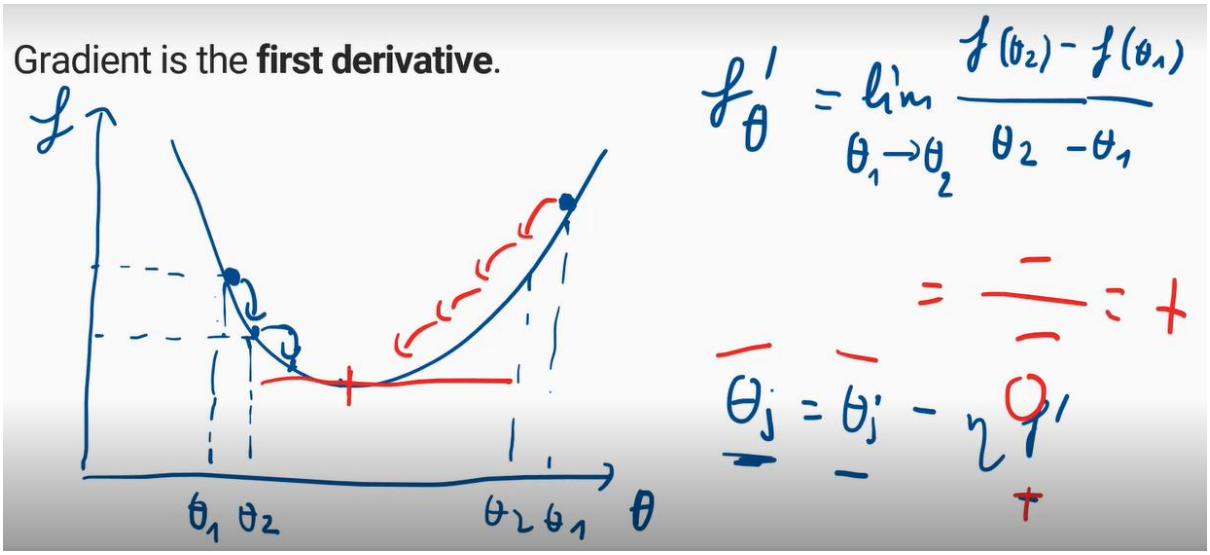
In more concrete terms, we begin by initializing with random values and populating the variable with them. The algorithm is then steadily improved, one tiny step at a time, with each effort to reduce the cost function (such as the MSE), until it converges to a minimum.



Slide in the class

➔ We can see that no matrix inversion: much faster than Normal Equation.

Direction for parameter's update



Slide in the class

To prove that its direction is always downward (minimum point) regardless of which side it starts, we rely on the definition of the derivative.

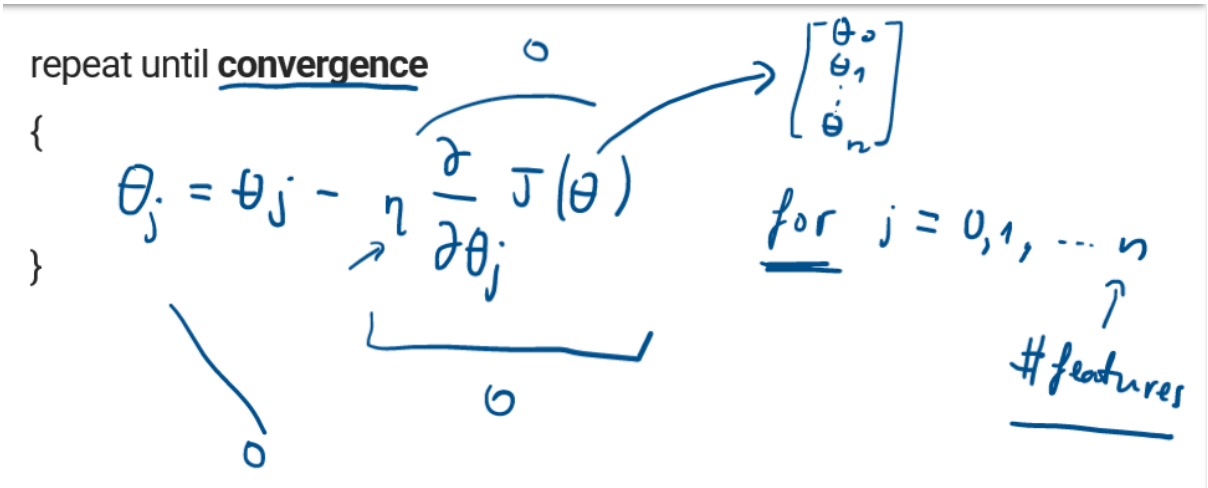
If starting from the left, its derivative calculates the new theta = the old theta + a small amount → pointing down to the minimum point. (Details are in the picture above).

And vice versa if starting from the right, its derivative will be calculated according to the new theta = the old theta – a small amount of → still points down to the minimum point (Details see on the image above).

Note: If we start at the minimum point, it will stay the same and not update theta anymore.

1.2.1. Batch Gradient Descent

Batch Gradient Descent algorithm



Slide in the class

We must calculate the gradient of the cost function with respect to each model parameter before you can use Gradient Descent. In other words, we need to figure out how much changing just a little will affect the cost function. Partially derivative describes this.

Equation 4-5. Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^\intercal \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

We compute BGD

Gradient vector

$$\nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

Slide in the class

Implementation of BGD using Gradient vector

Simply move in the opposite direction to go downhill once you get the gradient vector, which points uphill. This entails deducting MSE(from). The learning rate is important in this situation: To calculate the magnitude of the downhill step, multiply the gradient vector by.

Equation 4-7. Gradient Descent step

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

repeat until **convergence**

{

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathrm{MSE}(\boldsymbol{\theta})$$

}

Slide in the class.

Let's look at a quick implementation of this algorithm:

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

And we got the result:

```
>>> theta
array([[4.21509616],
       [2.77011339]])
```

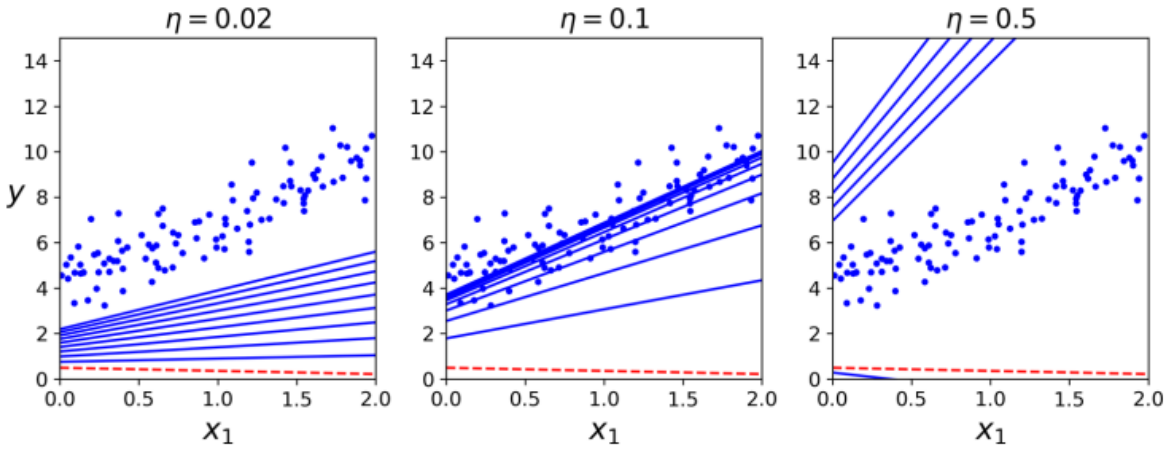


Figure 4-8. Gradient Descent with various learning rates

The algorithm will ultimately arrive at the solution on the left, but it will take a very long time because the learning rate is too low. The learning rate is most impressive in the middle, where it has already reached the solution after only a few iterations. The algorithm diverges, leaping all over the place, and really moving further and further away from the solution at each step on the right because the learning rate is too high.

Learning rate

The learning rate hyperparameter, which determines the step size in Gradient Descent, is a crucial parameter. The technique will take a long time to converge if the learning rate is too low and will require many iterations.

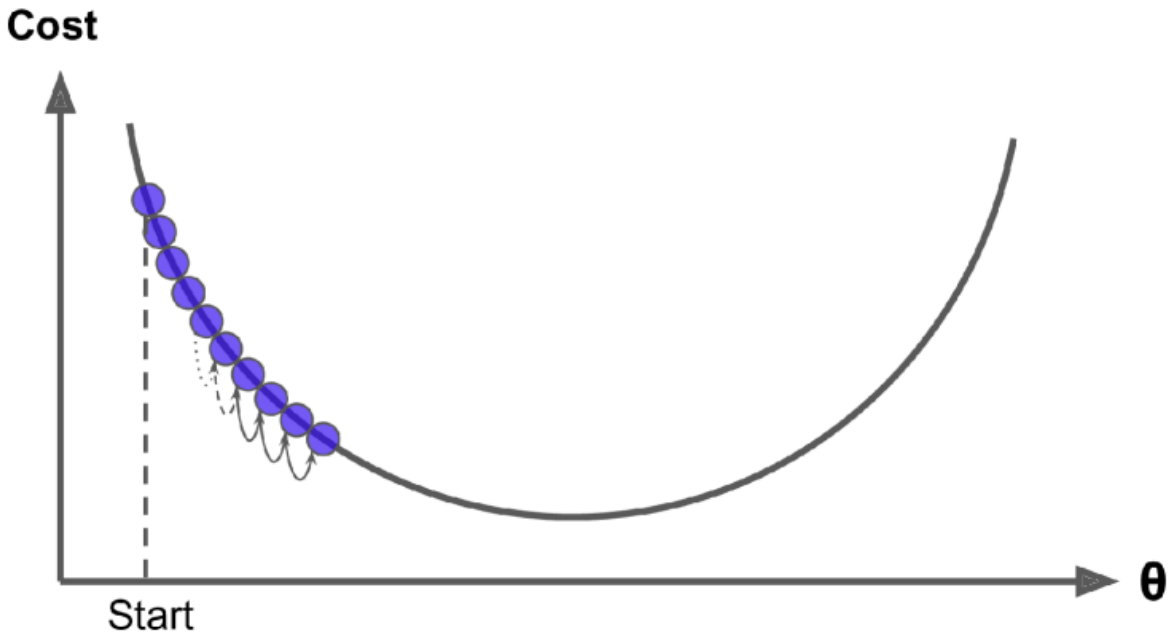
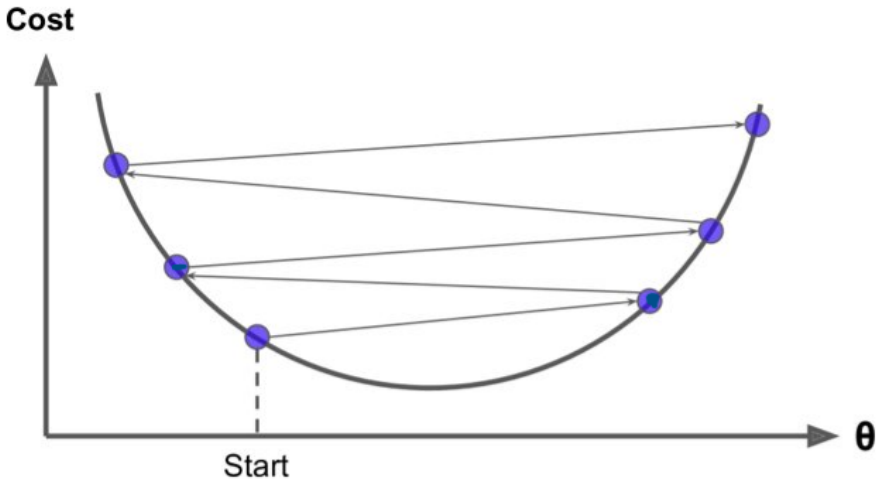


Figure 4-4. The learning rate is too small

On the other hand, if the learning rate is too great, you might jump across the valley and come out on the other side, perhaps higher than you started. This could cause the algorithm to diverge, using increasingly huge values and failing to arrive at a satisfactory answer.

Large ➔ may **no convergence**



The learning rate is too large

Problems of Gradient descent

Local minimal:

when the derivative is 0, it will stop and in many cases we may not find the extreme at which the MSE value is smallest.

Solution: Re-initialize the start value and rerun the algorithm several times, or use stochastic gradient descent.

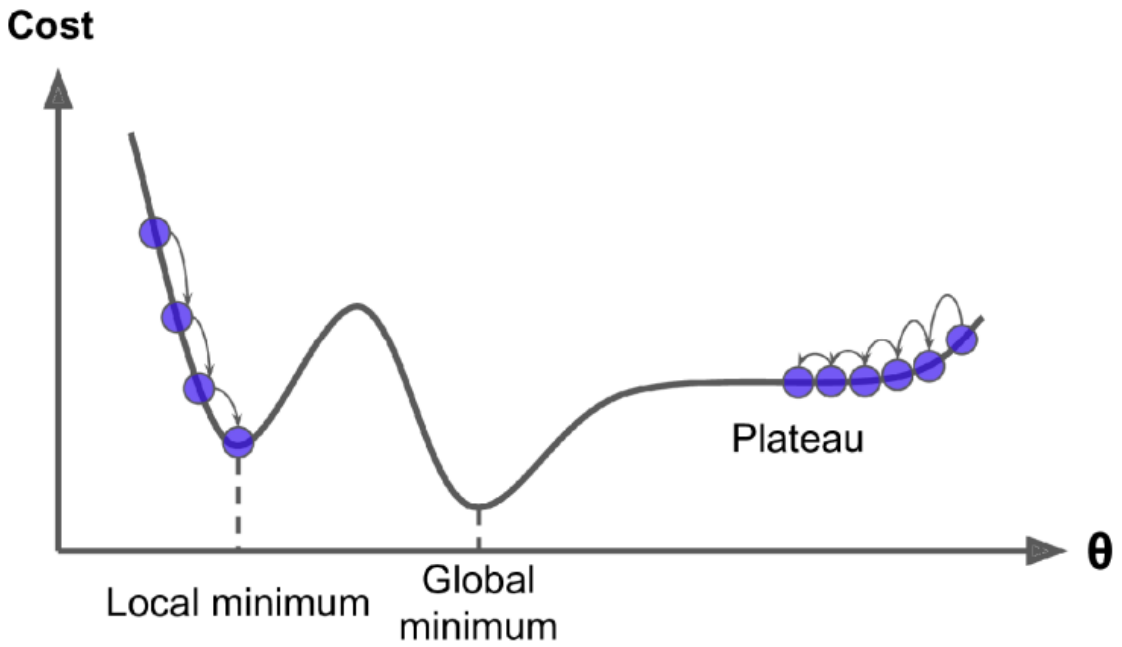
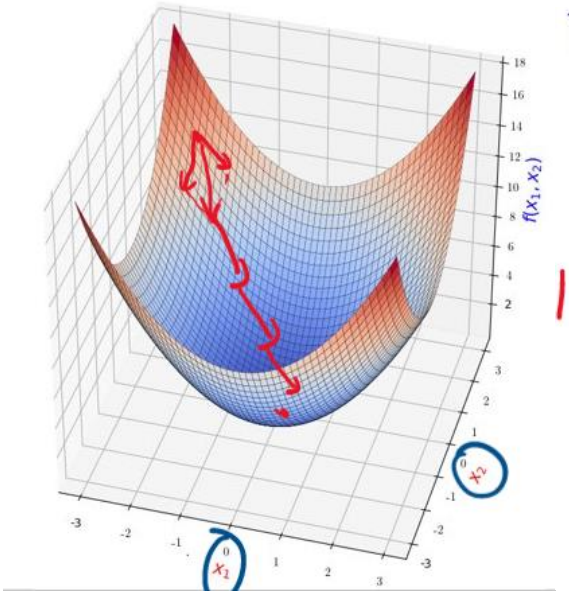


Figure 4-6. Gradient Descent pitfalls

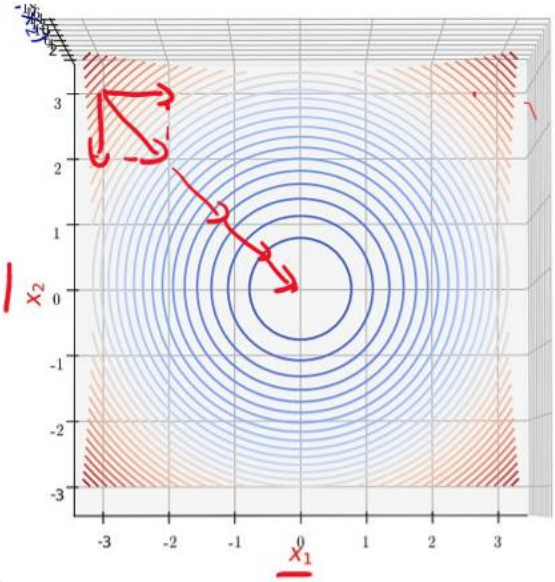
Features scales

When we take the derivative of two features with much different values, the vector will be misdirected and may cause errors.

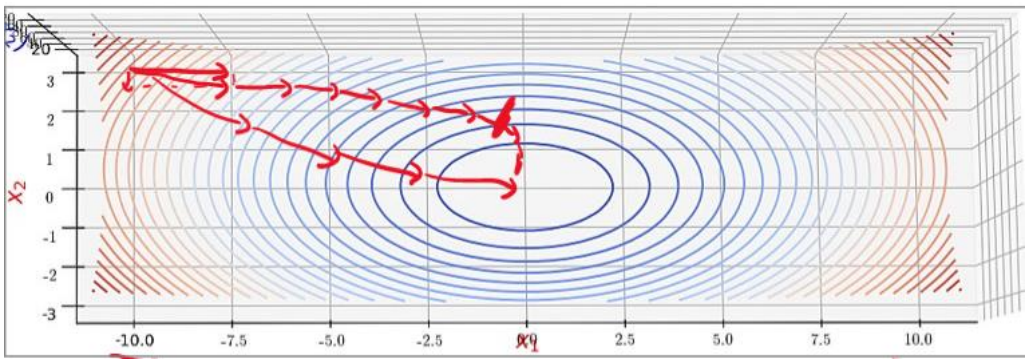
How to fix: Make the data on the features uniform, can divide all values in that feature by the maximum value



Similar Scales.



Similar Scales (Contour plot).



(Very) Different scales (contour plot).

1.2.2. Stochastic Gradient Descent.

BGD vs SGD

The fundamental issue with Batch Gradient Descent is that method is extremely sluggish when the training set is large since it computes the gradients for every step using the whole training set. On the other hand, Stochastic Gradient Descent computes the gradients based solely on one random instance from the training set at each step.

However, compared to Batch Gradient Descent, this approach is significantly less regular because of its stochastic (i.e., random) nature. The cost function will bounce up and down instead of steadily dropping until it hits the minimum.

Batch GD is **terribly slow** on very large training sets:

$$\frac{\partial}{\partial \theta_j}\text{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}\boldsymbol{\theta} - y^{(i)}\right)x_j^{(i)}$$

SGD uses only **one (random) sample** to compute gradients:

$$\frac{\partial}{\partial \theta_j}\text{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\mathbf{x}^{(i)}\boldsymbol{\theta} - y^{(i)}\right)x_j^{(i)}$$

Slide in the class.

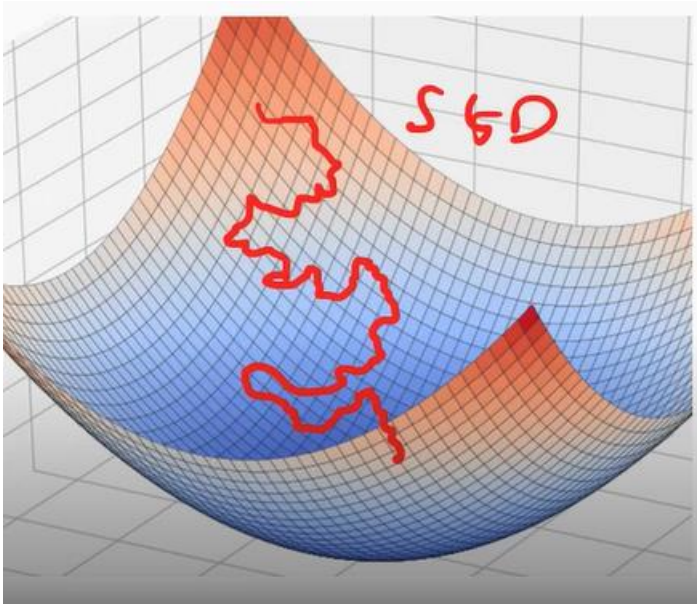
Advantage of SGD:

+ Much faster than BGD.

+ Uses much less memory than BGD.

Properties of SGD:

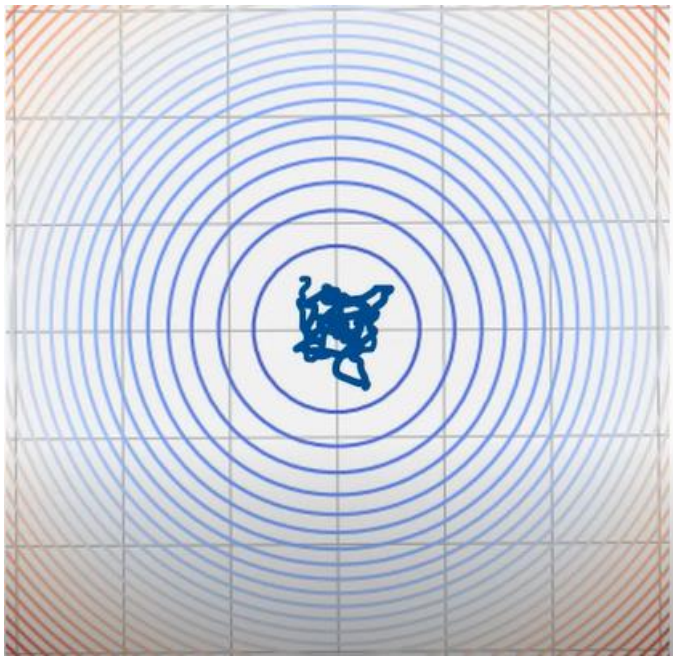
+ Run "Randomly": not decreases cost function on every iteration. (SGD's path is always complicated and zigzag, not straight like BGD. The cost function of SGD sometimes decreases and sometimes increases) → Thus can solve the problem of local minimal.



Slide in the class.

+ When at (closed to) the minimum: continue to bounce around.

In order for SDG to stop at the point we need to find, we have 2 ways that we can assign the number of loops available, or assign it a function with a decreasing value through each loop (to the value 0 then stop)



Slide in the class.

Implementation of SGD

To decrease learning rate properly, we create a function called **learning schedule**.

The learning schedule is the formula that calculates the learning rate for each iteration. We can find us trapped halfway to the minimum or locked in a local minimum if the learning rate is dropped too soon. If we stop training too soon after the learning rate has been dropped too slowly, we might hop around the minimum for a very long time and come up with a poor solution.

This code implements Stochastic Gradient Descent using a simple learning schedule:

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

And then we had the result

```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

Next, we will shows the first 20 steps of training:

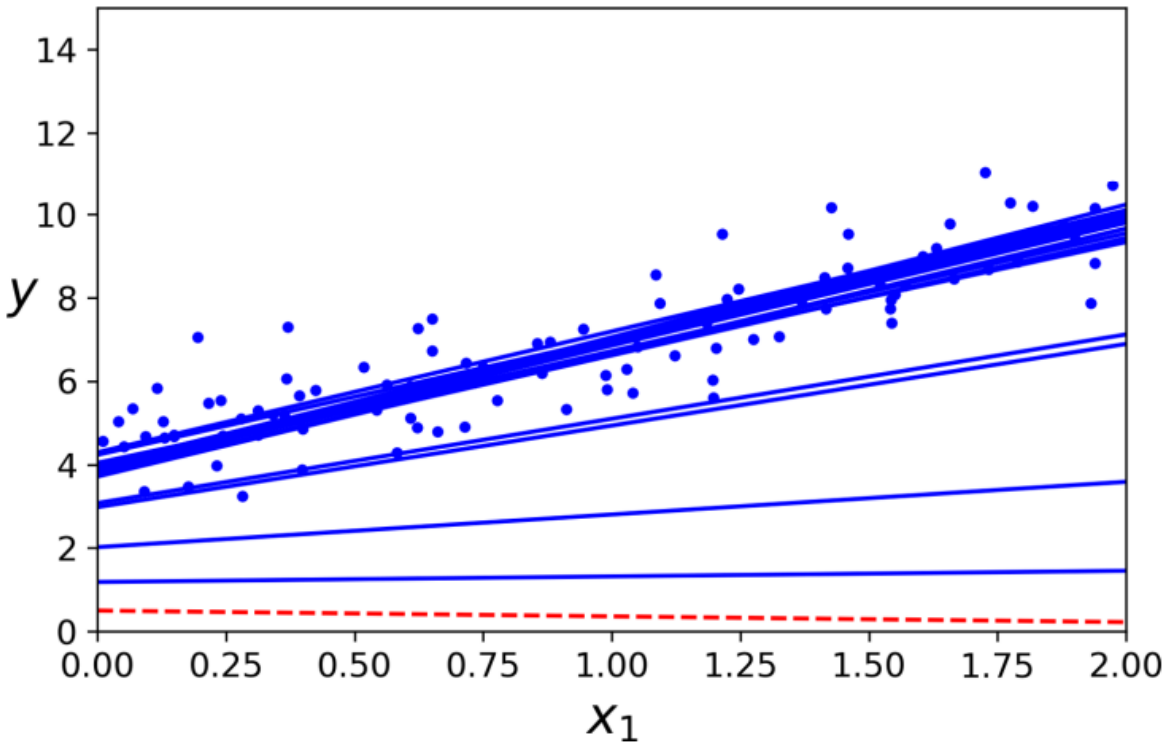


Figure 4-10. The first 20 steps of Stochastic Gradient Descent

Use Scikit-SGDRegressor Learn's class, which by default optimizes the squared error cost function, to perform linear regression using stochastic GD. The following program runs until the loss decreases by less than 0.001 during the course of one epoch, or for a maximum of 1,000 epochs (max iter=1000, tol=1e-3).

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Once again, we find a solution quite close to the one returned by the Normal Equation:

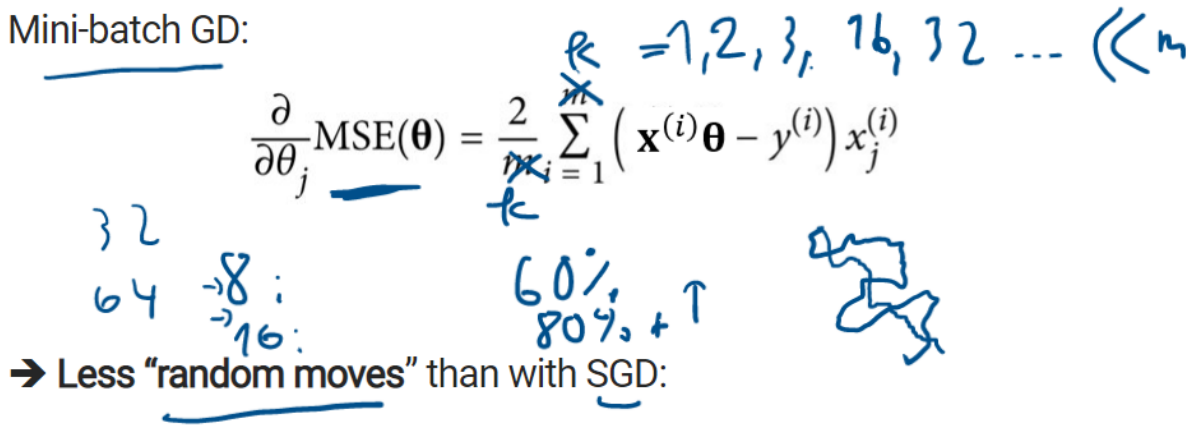
```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

1.2.3. Mini-batch Gradient Descent.

Once we are familiar with batch and stochastic gradient descent, it is easy to understand. Mini-batch GD computes the gradients on small random sets of instances called mini-batches at each step rather than using the entire training set (as in batch GD) or just one instance (as in stochastic GD).

Advantages:

The key benefit of adopting Mini-batch GD over Stochastic GD is that matrix operations can be hardware optimized for improved efficiency, especially when using GPUs.



Slide in the class.

Implement Mini-batch GD:

```
n_epochs = 50
minibatch_size = 20
theta = theta_random_init  # random
theta_path_mgd = [theta_random_init]
```

1.2.4. Comparison of LR training algorithms

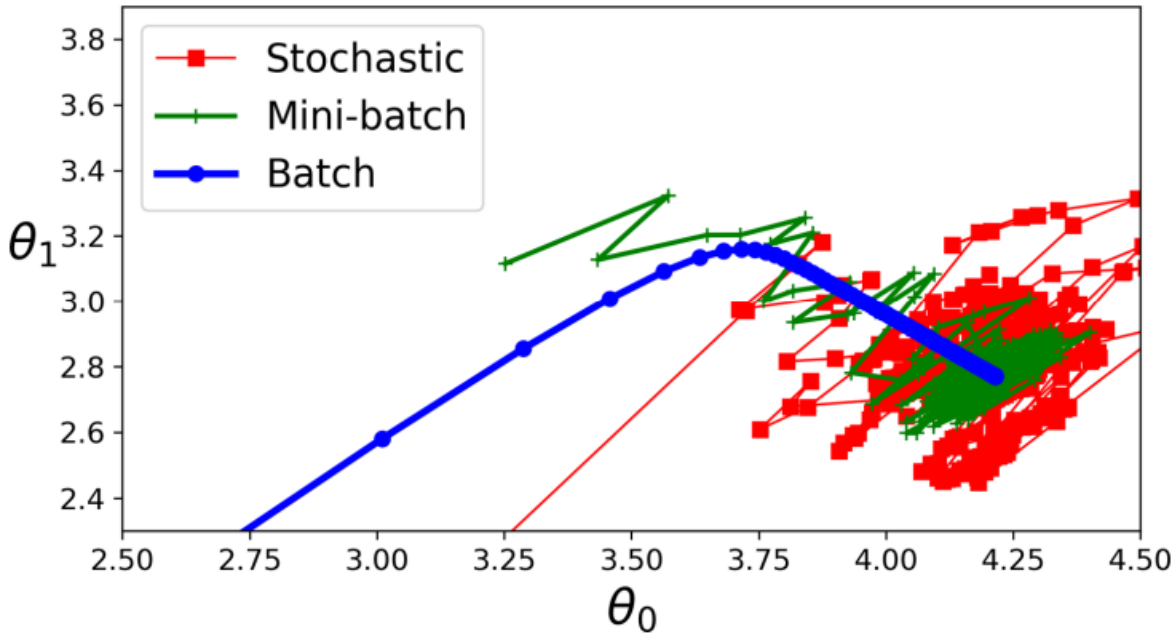


Figure 4-11. Gradient Descent paths in parameter space

Table 4-1. Comparison of algorithms for Linear Regression

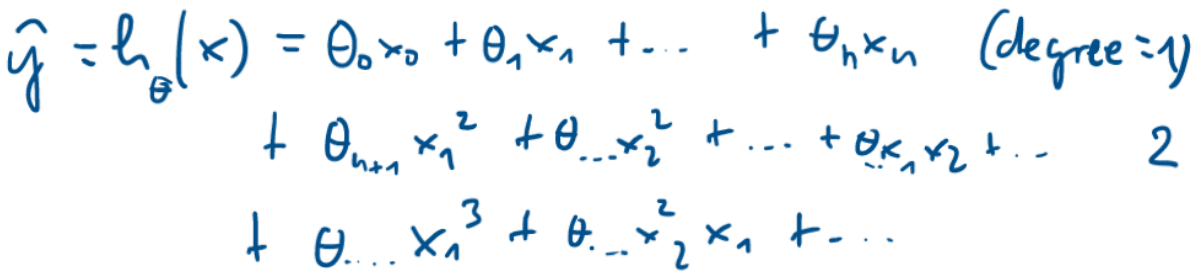
Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	N/A
SVD	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	`SGDRegressor`
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`

2. Polynomial Regression

What happens if your data isn't as simple as a straight line? Unexpectedly, you can fit nonlinear data using a linear model. A straightforward method for doing this is to train a linear model using the increased set of features after adding the powers of each feature as new features. The name of this method is polynomial regression.

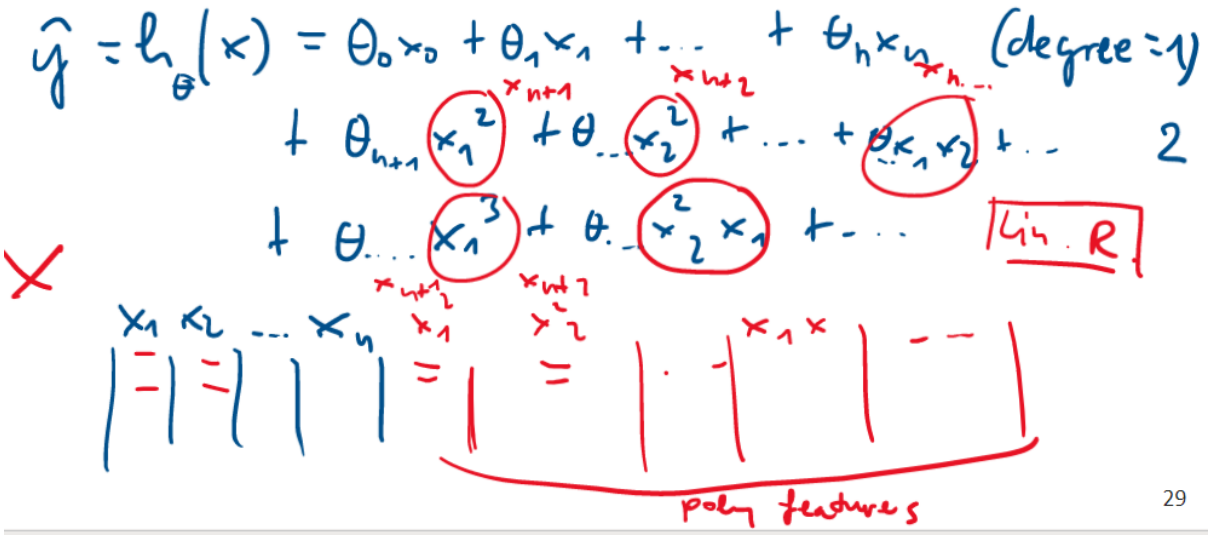
2.1. The hypothesis function of LR

Hypothesis equation can have many degrees (from order 1 -> order n).



Slide in the class.

We can transform the function above with another form of function that includes only order 1 by assigning features with higher orders in word order (n, n+1, n+2,...). This will help the function to have a level 1 that is easier to solve but has more features



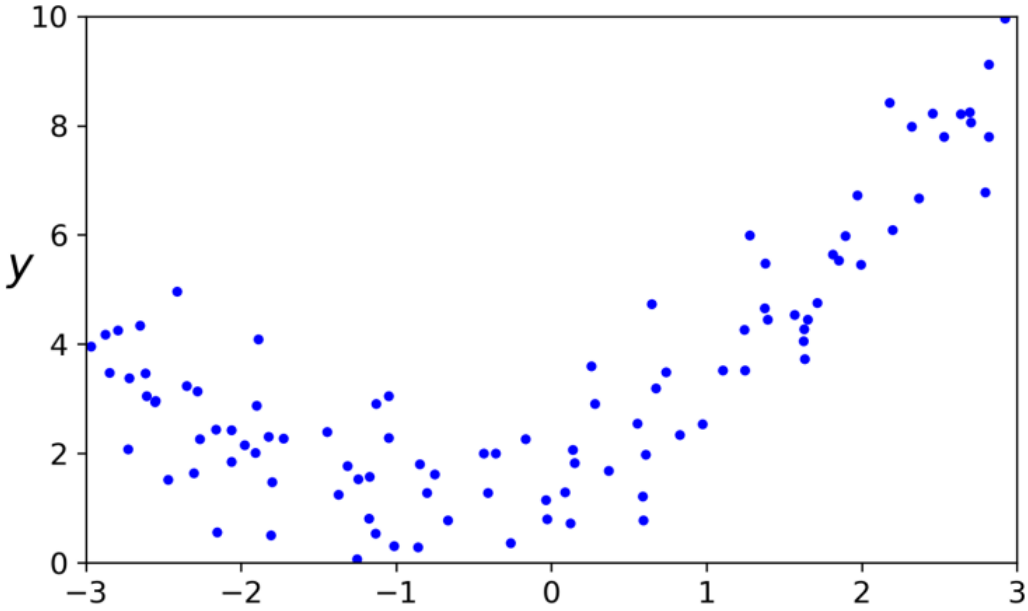
Slide in the class.

2.2. Training PR models.

We first generated nonlinear and noisy dataset.

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

And we got the result:



+ Add "new" features (high-order features).

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

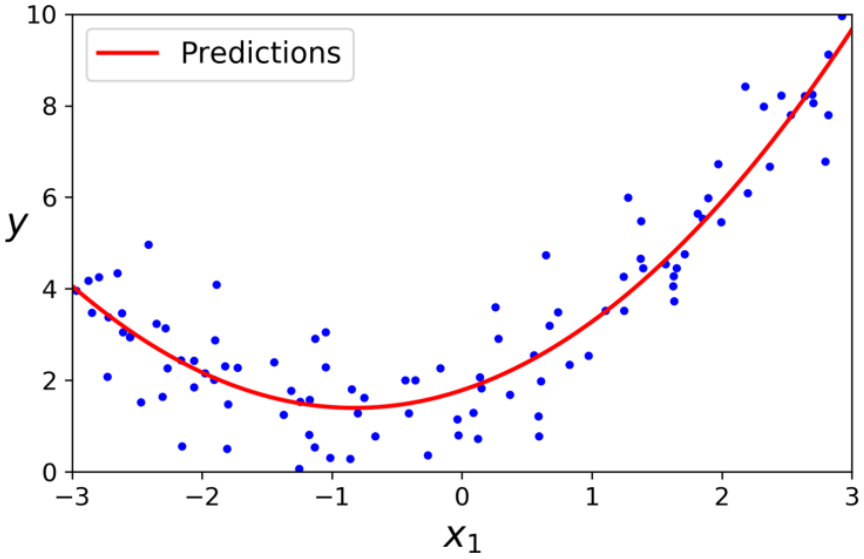
+ Run linear regression training algorithm (eg., Normal Equation, Gradient Descent).

X_poly now contains the original feature of X plus the square of this feature.

➔ we can fit a LinearRegression model to this extended training data.

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

And then plot the chart



Polynomial Regression model predictions.

2.3. Learning curves.

2.3.1. Under-fitting and Over-fitting.

Under-fitting: When our model is too simple for the data (for example, we have a quadratic function but only describe the data with a quadratic function).

Over-fitting: When our models are too complex for the data (for example, we have a quadratic function but use a quadratic function to describe the data).

2.3.2. Method to identify under-fitting and over-fitting.

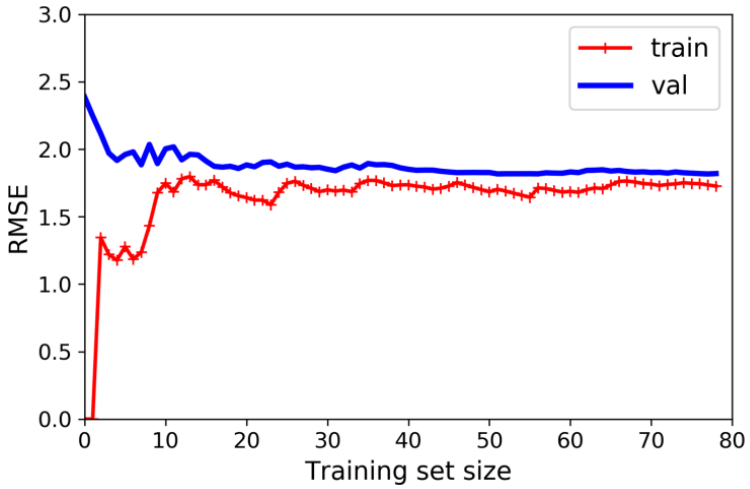
+ Use cross-validation:

- ✓ **Under-fitting**: If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting.
- ✓ **Over-fitting**: If it performs poorly on both, then it is underfitting.

+ Use learning curves: plot performance against training set size (or the training iteration). Train the model multiple times on various sized subsets of the training data to produce the graphs.

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

We got:



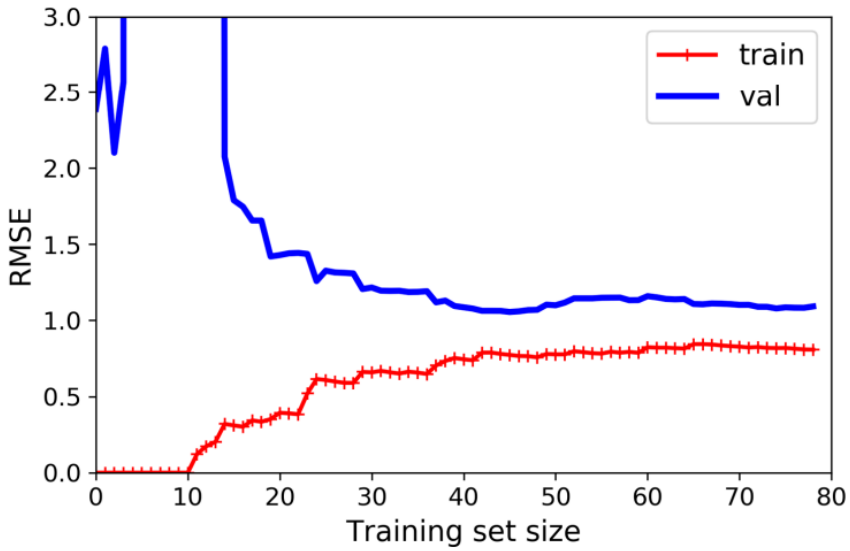
Learning curves.

We can see that the graph representing the training set and the validation set has a large MSE, which means that both give poor results, so it can be concluded that this case is **under-fitting**.

Next, we plot learning curves of a 10th-degree polynomial model on the same data:

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
```

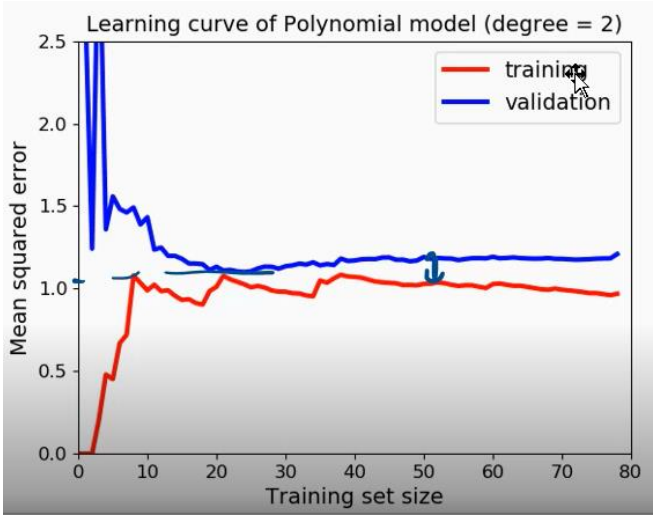
We got:



Learning curves for the 10th-degree polynomial model

We can see that: The graph representing the training set and the validation set has a large MSE and the distance between the two graphs looking along the MSE axis is also large. That means it is good in the training set but bad in the validator set, so it can be concluded that this is **over-fitting.**

Note: A special case if the MSE of both sets is small and the distance between the two lines is close, it is good-fitting.



CHAPTER 5: REGULARIZATION

1. To overcome Overfitting

1. Use more data
2. Use simpler models
3. Do regularization
4. Early stopping

2. Regularization

2.1. Idea

- Use a complex model: constrain the model, and fit the pull-down model complexity to the data. We have 3 regularization methods:

- Ridge regression
- Lasso regression
- Elastic net

2.2. Ridge regression

- *Cost function*:

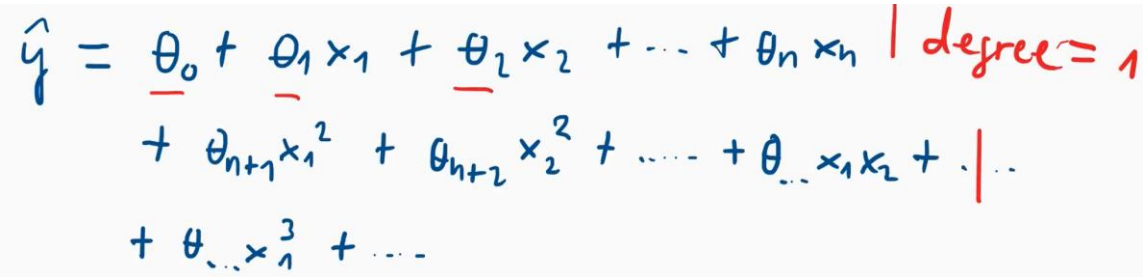
Ridge regression : regularization term

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Alpha: number more than 0 (larger alpha => stronger regularization => smaller thetas. Alpha used to control the strength of regularization).

Thetas: the parameters of the model.

- When weights are small:



+ If the equation is of order 1 then all parameters with degree greater than 1, theta will be zero, same with greater order function. The higher the tier, the more complicated it will be.

- *Proceeding*: when training the data, we want the cost function to be small. Therefore, the MSE and regularization term must be small. Therefore, when we put the regularization term in the algorithm, it will try to make the cost function small and make its absolute value smaller. When the data is a high-order function, the graph will fluctuate very strongly, but when the value of the parameters of the function is small, the function will fluctuate less, the smaller the parameter, the less the function will fluctuate. It will help us to describe the data better.

- *Notes*:

+ The cost function for training can be different from the performance measure of the model.

+ Feature scaling is important for regularized models to work.

– *Code:*

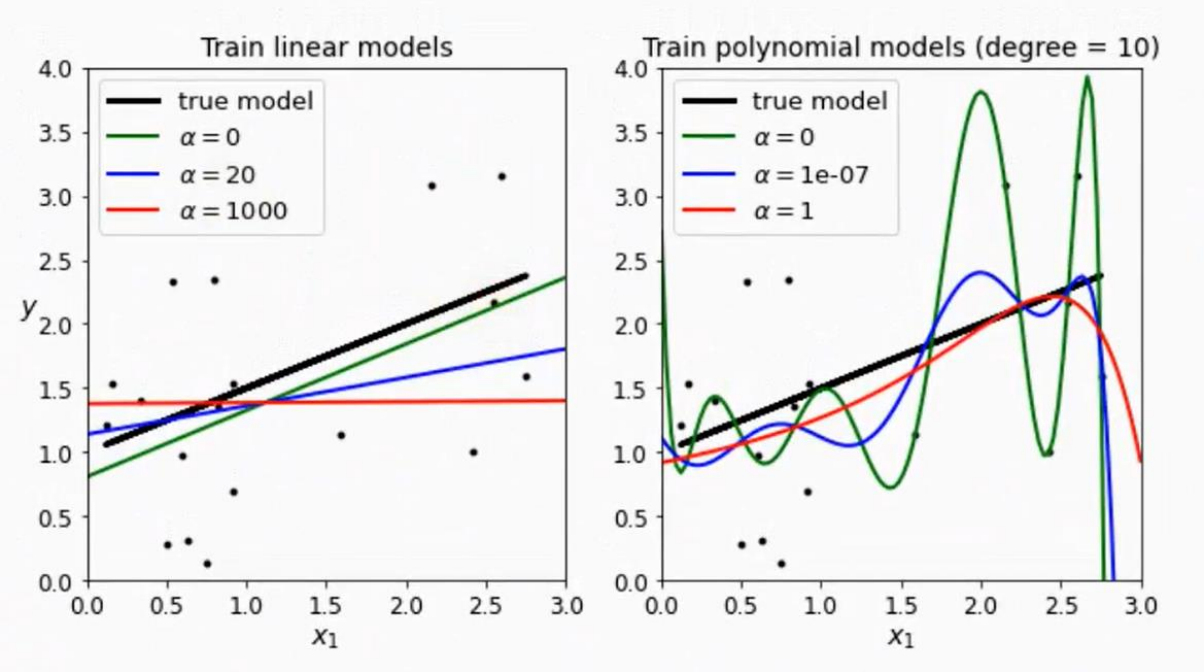
1. Generate linear data:

```
# 8.2. Ridge regularization
# 8.2.1. Generate linear data
m = 50
np.random.seed(15);
X = 3*np.random.rand(m, 1)
y_no_noise = 0.7 + 0.7*X
y = y_no_noise + np.random.randn(m, 1)/1.5
X_test = np.linspace(0, 3, 100).reshape(100, 1)
```

2. In scikit learn already has ridge regression function for training model. We only have to declare it and use. (we will set alpha = 1)

```
# 8.2.2. Train a ridge model and predict
#ridge_reg = Ridge(solver = 'cholesky', alpha=1, random_state=42) # train using closed-form solution
ridge_reg = Ridge(solver = 'sag', alpha=1, random_state=42) # train using stochastic GD
#sgd_reg = SGDRegressor(penalty="l2") # train using stochastic GD: 'l2' norm is ridge regularization
ridge_reg.fit(X, y)
ridge_reg.predict([[1.5]])
```

3. Plot models: We will try with increasing alpha values (0, 20, 100).
4. After plot:



- Degree = 1: our model is fitting the data. With alpha = 20, the blue line is deviated from the true line. Continue to increase alpha = 1000, the red line is already horizontal because when alpha is too large the thetas prices are almost zero and will be equal to thetas zero
- Degree = 10: Overfitting. With alpha =0, the graph will fluctuate a lot (green line). With alpha = 10^(-7), the graph will still fluctuate strongly but less than the green line (blue line). With alpha = 1, the graph will be pulled to good-fitting due to less oscillation, almost straight line (red line).

2.3. Lasso regression:

- *Cost function:*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \Sigma_{i=1}^{n} |\theta_i|$$

- *Compare Lasso and Ridge regression*: With the same alpha, Lasso will give the lower thetas because normal thetas will be smaller than 1. In Ridge, when we square the thetas will get smaller, the alpha itself is already reduced by squaring so we don't need to narrow the thetas smaller. In lasso, if we want our sum to be small, we must be small, alpha > 0, so thetas must be small, lasso will pull small thetas values stronger than Ridge.

=> Lasso is stronger than Ridge

- *Code:* Same with Ridge, it also already have library support for us.

```
from sklearn.linear_model import Lasso

lasso_reg = Lasso()
lasso_reg.fit(X,y)
lasso_reg.predict([[1.5]])
```

2.4. Elastic net:

- *Elastic net is a combination of Ridge and Lasso regression*

- *Cost function:*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r\alpha \Sigma_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \Sigma_{i=1}^{n} \theta_i^2$$

α_1 l_1 α_2 l_2

- With r = 0, it will return Ridge regression(l2)

- With r = 1, it will return Lasso regression(l1)

- With $0 < r < 1$, combination of 2 above .

- *Code:* Same with Ridge, it also already have library support for us.

```
from sklearn.linear_model import ElasticNet

elastic_reg = ElasticNet(l1_ratio = 0.9)
elastic_reg.fit(X,y)
elastic_reg.predict([[1.5]])
```

2.5. Which one to use:

- Listed in order of preference (in practice):

1. Non-regularization (least preferred): should limit use, we should only use it when we know for sure what degree our model is.
2. Ridge regularization.
3. Lasso regularization.
4. Elastic net (most preferred): should be used the most.

CHAPTER 6 EARLY STOPPING

1. Idea

- This algorithm is used to avoid overfitting due to long training.

2. Implementation

- With Stochastic and Mini-batch GD, the curves may be not so smooth.

3. Code

1. Generate:

```
# 9.4. Do early stopping
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1, tol=-np.inf, # tol<0: allow loss to increase
                       warm_start=True,  # warm_start=True: init fit() with result from previous run
                       penalty=None, learning_rate="constant", eta0=0.0005, random_state=42)
n_iter_wait = 200
minimum_val_error = np.inf
from copy import deepcopy
train_errors, val_errors = [], []

from sklearn.metrics import mean_squared_error
for iter in range(1000):
    # Train and compute val. error:
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    # Save the best model:
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_iter = iter
        best_model = deepcopy(sgd_reg)
    # Stop after n_iter_wait loops with no better val. error:
    if best_iter + n_iter_wait < iter :
        break

    # Save for plotting purpose:
    val_errors.append(val_error)
    y_train_predict = sgd_reg.predict(X_train_poly_scaled)
    train_errors.append(mean_squared_error(y_train, y_train_predict))
train_errors = np.sqrt(train_errors) # convert to RMSE
val_errors = np.sqrt(val_errors)
```

2. Plot model: loop around 250 will give the best results

CHAPTER 7: LOGISTIC REGRESSION (CLASSIFICATION)

1. What is logistic regression:

Logistic regression is a binary Classifier (classes: 0, 1).

- It outputs the probability p that a sample belongs to the positive class.

- ***Example:*** must predict whether the picture is a cat or not, will produce 2 results: cat(1) , not cat(0)

If p>0.5: belong to class 1

If p<0.5: belong to class 0

- Hypothesis function of logistic regression:

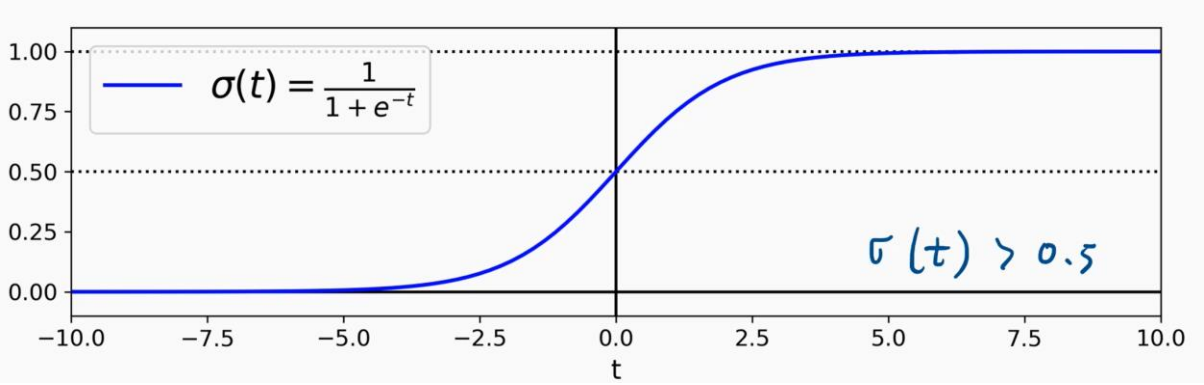
+ The predict probability:

$$\hat{p} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\mathbf{x}\boldsymbol{\theta})$$

+ Where σ is the logistic function:

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

2. Sigmoid function



If t = 0, σ = 0.5.

If t very large, σ will be very large asymptotically about 1.

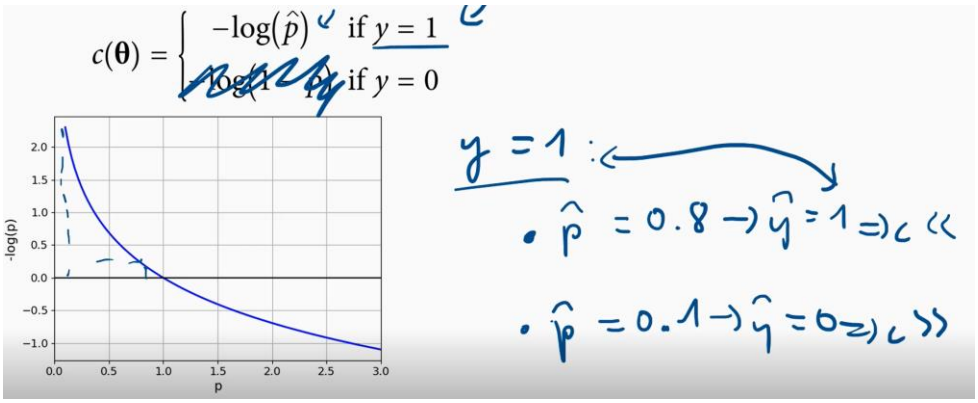
If t very small, σ will be very small asymptotically about 0.

3. Cost function

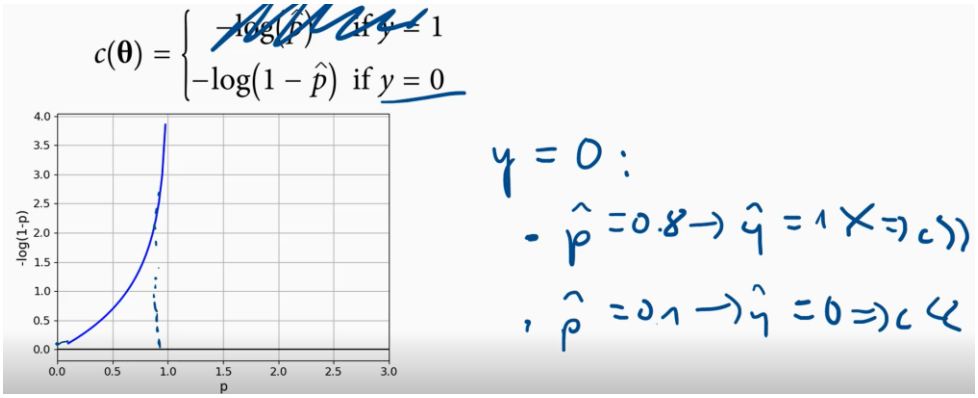
1. Cost function of 1 sample:

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

When y = 1



When y =0



2. Cost function of n sample: (is called the Log loss function)

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - \hat{p}^{(i)}\right)\right]$$

- Training with log loss function: no closed-form solution, log loss is convex.

- Training using Gradient descent:

$$\frac{\partial}{\partial\theta_j}J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\mathbf{x}^{(i)}\boldsymbol{\theta}\right) - y^{(i)}\right)x_j^{(i)}$$

- *Example: use predict Iris dataset*

+ 4 features: length of petal, width of petal, length of sepal and width of sepal. (we only get 1 feature for train data, we get width of petal)

+ 3 labels: Virginica, Versicolor and Setosa.

+ 150 samples

⇨ Use logistic regression model to train

4. Code

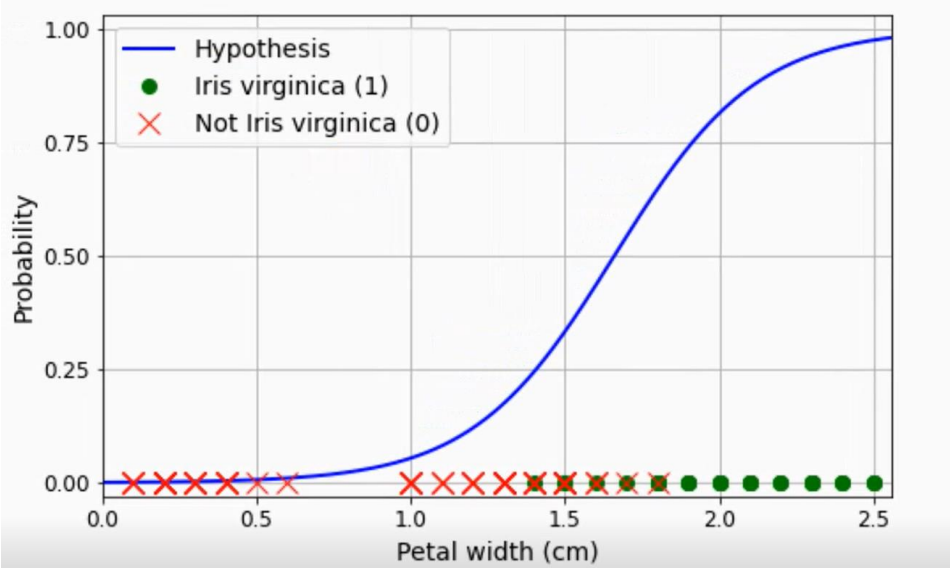
1. *Load data*

```
# 10.2. Load Iris dataset
from sklearn import datasets
iris = datasets.load_iris()
print(iris.keys()) # data: features, target: label
#print(iris.DESCR) # description of the data
```

2. *Train (True if Iris virginica a and False if not Iris virginica)*

```
#%% 10.3. Train a logistic regression model
X = iris["data"][:, 3:] # Use only 1 feature: petal width (max: 2.5 cm)
y = (iris["target"] == 2).astype(np.int) # 2 classes: True if Iris virginica, else False
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(solver="lbfgs", random_state=42)
log_reg.fit(X, y)
```

3. *Plot model*

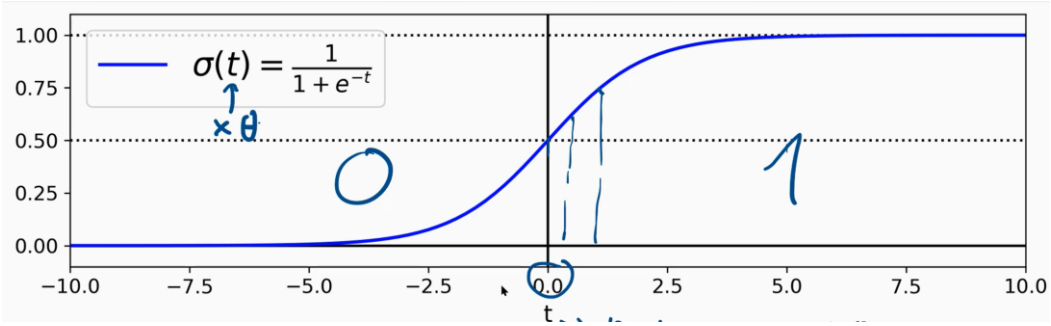


i. At the position petal width is 1.6, this point will split our feature into 2 parts (class 1 and 0), is called ***decision boundary***.

4. *Decision boundaries:*

$$\sigma(t) = \frac{1}{1+e^{-t}}$$

$\times \theta$



5. We have t = $t \rightarrow \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$

+ t >=0: p >= 0.5: class 1

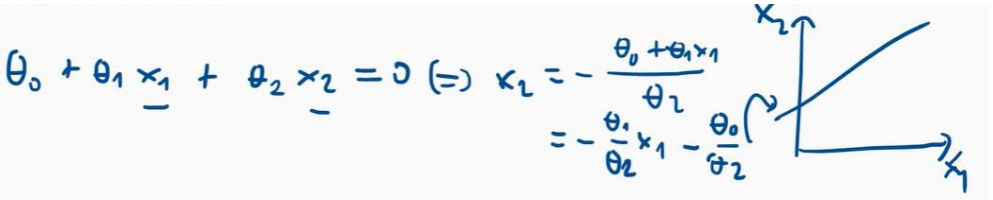
+ t = 0: decision boundary

+ t <=0: p<0.5: class 0

5. *Decision boundaries of linear models:*

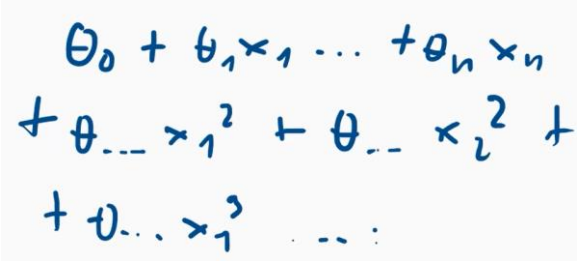
   a. For 1-feature data:

b. For 2-feature data:



6. *Non-linear decision boundaries:*

c. It has the form:



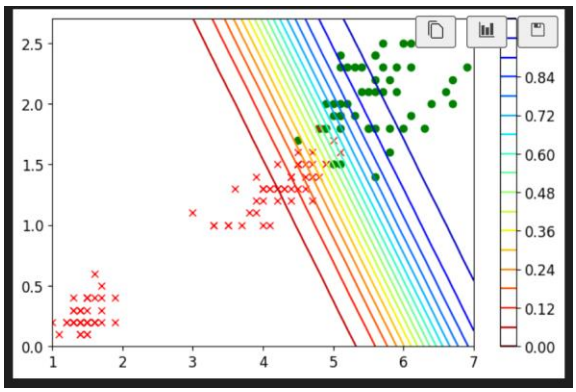
d. Code: we will use 2 features (petal length and petal width)

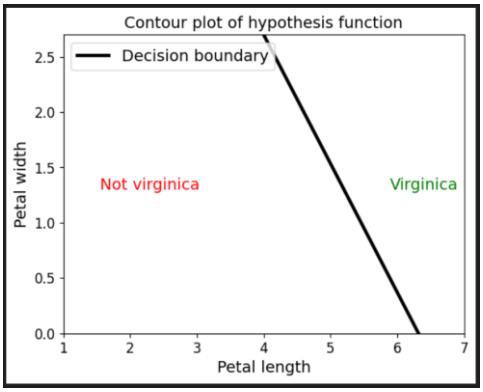
```
# 10.6. Train another model, using 2 features
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.int)  # 2 classes: True if
log_reg = LogisticRegression(solver="lbfgs", random_state=42)
log_reg.fit(X, y)
```

e. After run:

```
Decision boundary: x1 = [1.66041261]
```

f. Plot model:





CHAPTER 8: SOFTMAX REGRESSION (CLASSIFICATION)

1. What is Softmax Regression

- *Softmax regression* (a.k.a. Multinomial Logistic Regression) is generalization of logistic regression.

2. Hypothesis function of softmax regression

+ Given a classification of K classes, the hypothesis function of class k.

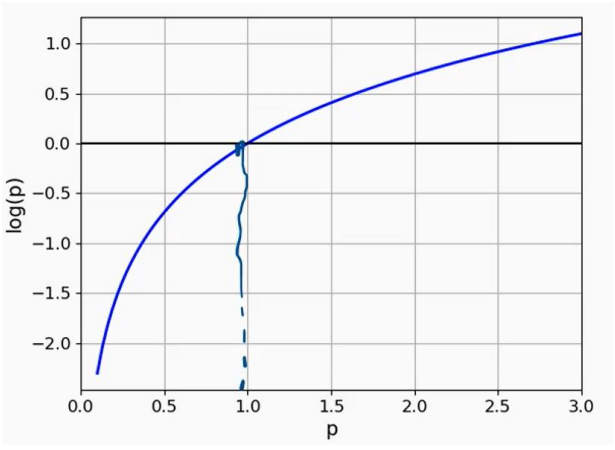
$$\hat{p}_k = h_{\boldsymbol{\theta}^{(k)}}(\mathbf{x}) = \frac{\exp(\mathbf{x}\boldsymbol{\theta}^{(k)})}{\sum_{j=1}^{K} \exp(\mathbf{x}\boldsymbol{\theta}^{(k)})}$$

3. Prediction class

$$\hat{y} = \underset{k}{\operatorname{argmax}}\,(p_k)$$

4. Cost function of softmax regression(is called the Cross entropy function)

$$J(\boldsymbol{\Theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$



g. When we have y = 0 ⇔ y_k=1, y_j=0 ∀ j ≠ R

h. y = k (true predict) ⇔ P_k = 0.8 => J ≈ 0 (true).

i. y ≠ k (wrong predict) ⇔ P_k = 0.1 => J » (wrong).

5. Gradients of the cross entropy cost functio

- We can use gradients by derivating the cost function of softmax. The gradient vector of the cross entropy cost function with regards to parameters $\Theta^{(k)}$ class k:

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

⇨ Now you can use Gradient descent to find parameters that minimize the cost function.

6. Code

- We will train Iris model and use LogisticRegression and use SoftMax regression using multi_class="multinomial" which means using Softmax Regression algorithm.

```
# 11.2. Train a softmax model for Iris data
X = iris["data"][:, (2,3)]   # petal length, petal width
y = iris["target"]  # use all 3 classes
softmax_reg = LogisticRegression(multi_class="multinomial", # multinomial: use Softmax regression
                                 solver="lbfgs", random_state=42) # C=10
softmax_reg.fit(X, y)
```

+ We try to predict the sample 126.

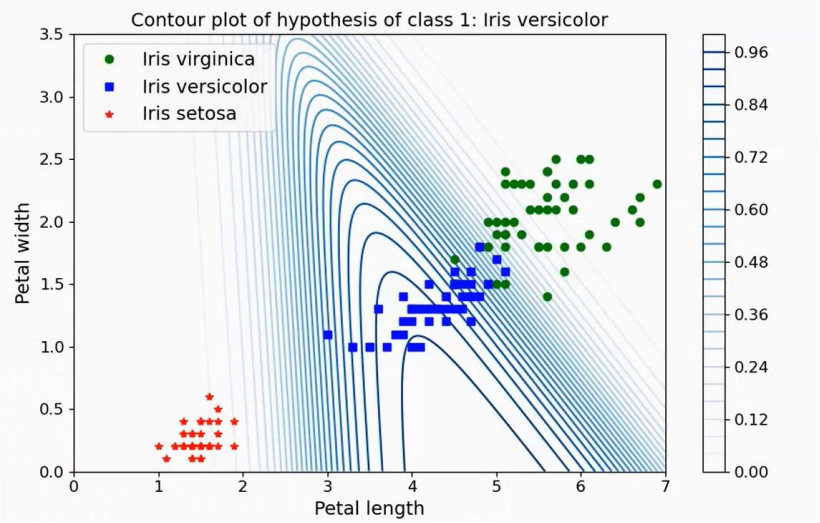
```
# 11.3. Try prediction
sample_id = 126
print(softmax_reg.predict_proba([X[sample_id]]))
print(softmax_reg.predict([X[sample_id]]))
print(y[sample_id])
```

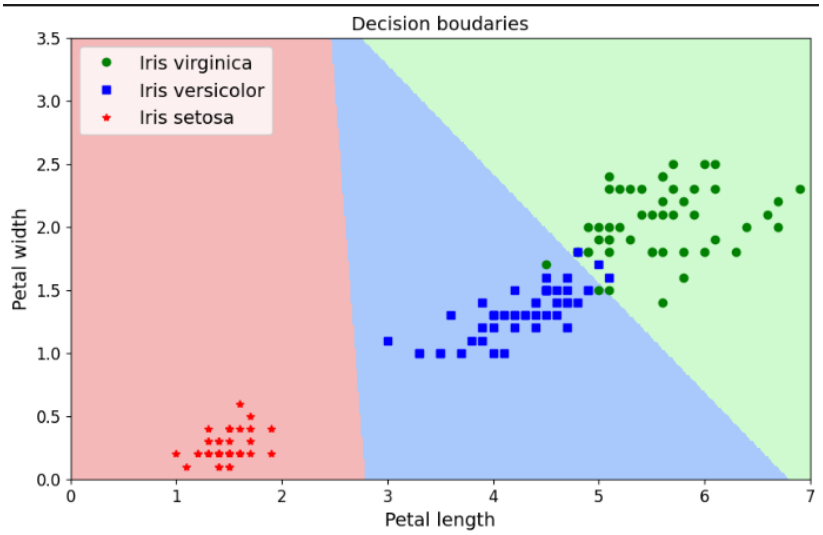
```
[[0.00095563 0.45412437 0.54492   ]]
[2]
2
```

➔ We can see it will call the argmax function and return the Prediction(line 2) and line 3 is the true label.

+ Plot model:

- Hypothesis for class 1 (Iris versicolor)


Contour plot of hypothesis of class 1: Iris versicolor
Iris virginica
Iris versicolor
Iris setosa
Petal width
Petal length
0.96
0.84
0.72
0.60
0.48
0.36
0.24
0.12
0.00


Decision boudaries
Iris virginica
Iris versicolor
Iris setosa
Petal width
Petal length

CHAPTER 9: SUPPORT VECTOR MACHINE(SVM)

1. Introduction:

- Support Vector Machine ***is the most popular models*** in machine learning.

- A Support Vector Machine (SVM) is a very powerful and versatile ML model.

- We can use SVM to do many things:

    + Classification: linear and non-linear.

    + Regression: linear and non-linear.

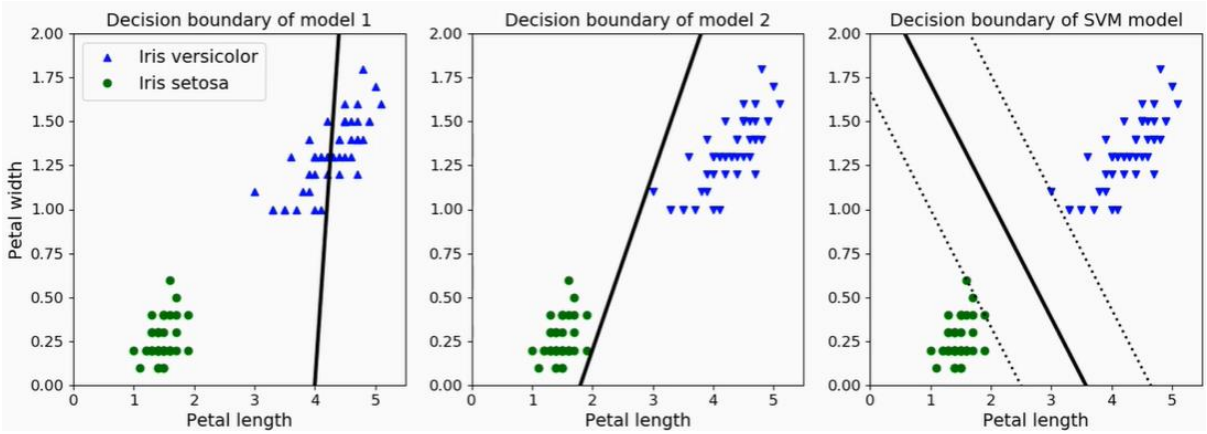
    + Clustering(unsupervise leaning), outliner detection.

2. Classification with linear SVM models:

2.1. Which decision boundary is good?

We will use 3 algorithms to compare efficiency:

+ SVM: binary classifier



⇨ Model 2 and SVM are good, model 1 is not good because it cannot classify 2 data. The SVM model is the best of the three models because it has an large margin so the data never crosses the decision boundary. SVM is the algorithm that finds the largest possible decision boundary, so it is the most accurate classification algorithm.

2.2. How SVM can find large decision boundary?

We will have 2 parallel lines (*support vector*), then drag 2 lines pull away from both sides to keep it apart, until it hits the first point of our class, it will stop. We will take the 2 lines just dragged as the 2 margins of the margin and we will measure again. Then we will rotate the 2 paths a bit and move the 2 paths away from each other until we touch the samples of the class and we will also measure the margin. Repeat(360 degree rotation) until find the maximum margin.

2.3. Hard margin and Soft margin.

- *Hard margin:* it will force all points belonging to a class to be on the same side

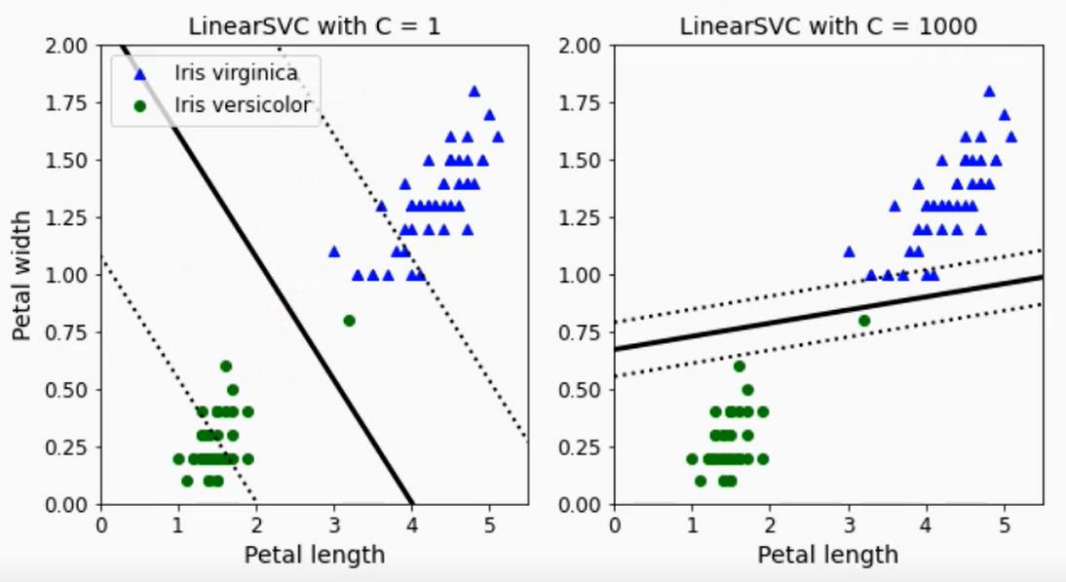
+ It will have 2 problems:

1. It requires classes to be discrete from each other

2. Since it drifts to the sides, it will be affected by points that are outside the edges. It will make the margin smaller.

- *Soft margin:* In contrast to hard margin, it will allow samples to lie on different sides. It allows the samples to exceed the margin so that it keeps the largest margin possible.

=> *Implementaion note:* when we set C = infinite, it is hard margin so when we need to use soft margin we only need to set C is low.



6. On the left *is soft margin* and on the right is *hard margin*.

=> SVM is an algorithm that tries to keep the margin as large as possible. SVM supports both hard margin and soft margin, normally will use soft margin to use because it is not affected much by outliners.

2.4. Classification with non-linear SVM

There are 2 methods to create a non-linear SVM.

+ Method 1: Add polynomial features

+ Method 2: Add similarity features

2.4.1 Add polynomial features.

- Similar idea to Polynomial regression.

+ Step 1: Add polynomial features.

+ Step 2: Linear regression

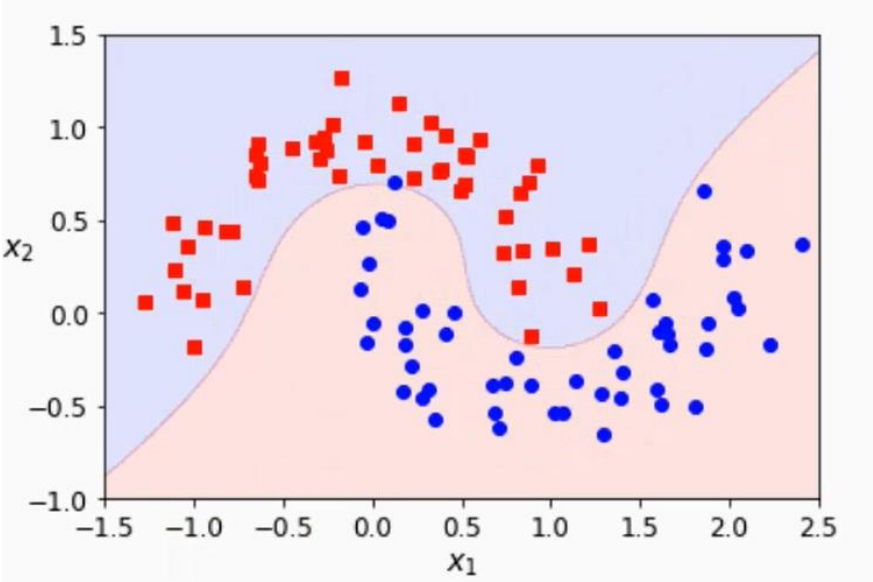
7. *Code*:

```
# 4.1. Add polinomial features and train linear svm
from sklearn.preprocessing import PolynomialFeatures
polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=40, random_state=42)) ])
polynomial_svm_clf.fit(X, y)
```

+ We add Polynomial Features have degree = 3.

+ Standard Scaler : help the program run stably.

+ We call “LinearSVC” and using polynomial features can draw decision boundary curve (we declare c = 40 to draw soft margin).



- To create a line with many curves, it will need large polynomial, when there is a large order, the higher order features will be added a lot, it will meet the same situation with polynomial regression and will make the algorithm run slow even memory overflow.

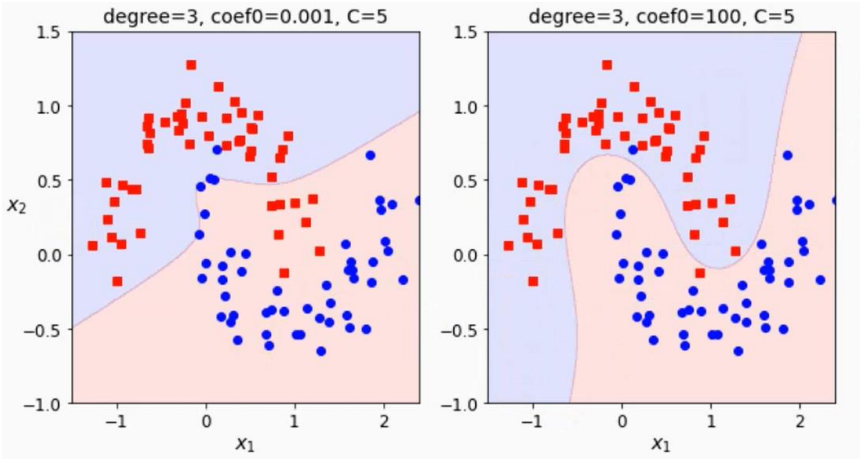
- In SVM, there is a method called ***Kernal Trick***, it is a trick that helps the algorithm to generate complex decision boundaries equivalent to higher degree polynomials but we don't need to add features with high degree.

- *Code:* ***Kernel trick*** is already installed in ***sklearn*** so we just need to declare and use.

```
#%% 4.2. Kernel trick for method 1: Polynomial kernel
from sklearn.svm import SVC
```

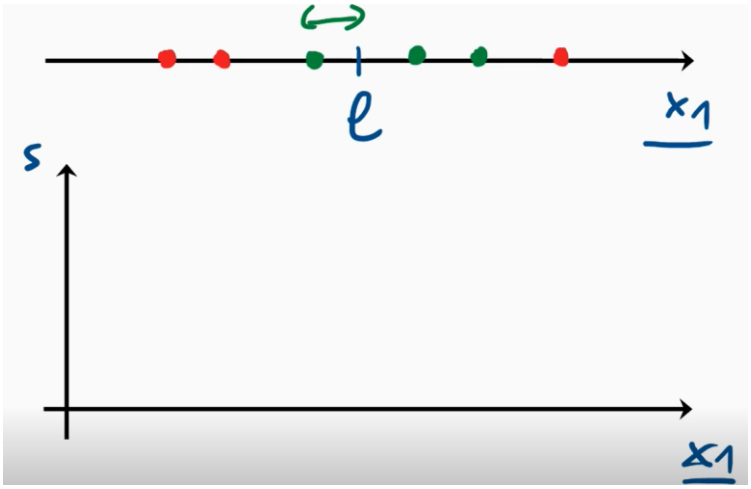
```
# NOTE:
#   larger coef0 => the more the model is influenced by high-degree polynomials
poly_svm_1 = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=.001, C=5))  ])
poly_svm_1.fit(X, y)

poly_svm_2 = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=100, C=5))  ])
poly_svm_2.fit(X, y)
```

+ The larger the coef0, the more complex the model will be.

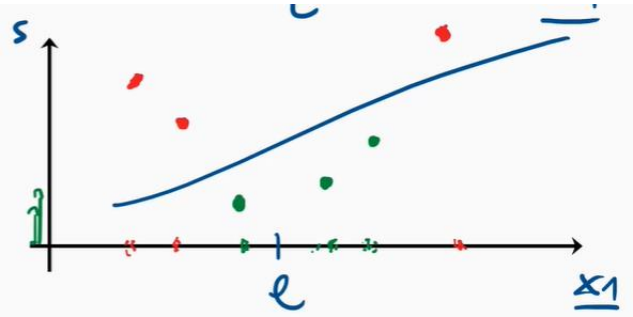


3. Add similarity features.

- When on the dataset there is only 1 feature (1D space): it will add feature to convert to 2D space.



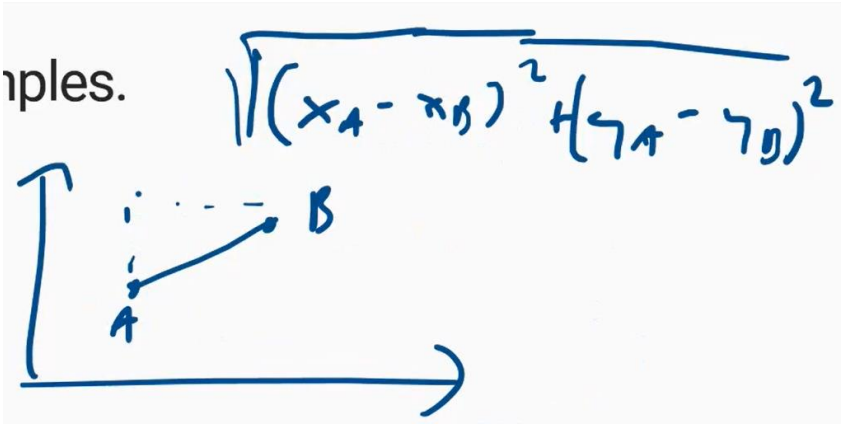
8. We will add a s feature, we will select a point (landmark) on our main feature, then the distance from sample to landmark will be feature s.
9. After plot:



1. **Similarity function**: Used to measure similarities b/w samples.

10. Typical similarity functions:

+ ***Euclidean distance***



+ ***Gaussian Radial Basis Function***: RBF function.

$$K_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

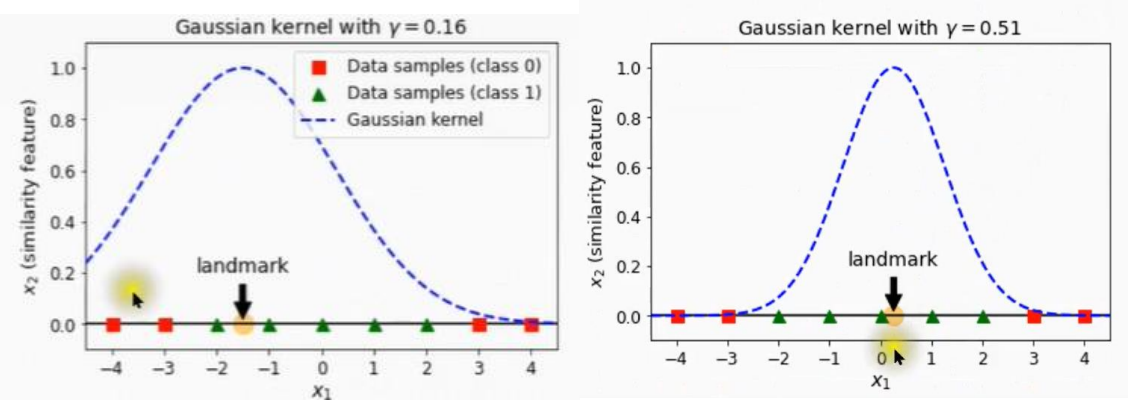
γ: coefficient of Gaussian Kernel.

11. Code:

1. Generate 1-feature data:

```
# 5.1. Generate 1-fearture data (1-dimenstional data)
X_1D = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4]).reshape(-1,1)
y = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0]) # 2 classes
```

2. We will plot 2 Gaussian kernel 1 and 2 (gamma = 0.16 and gamma = 0.51)



2. Data transformation:

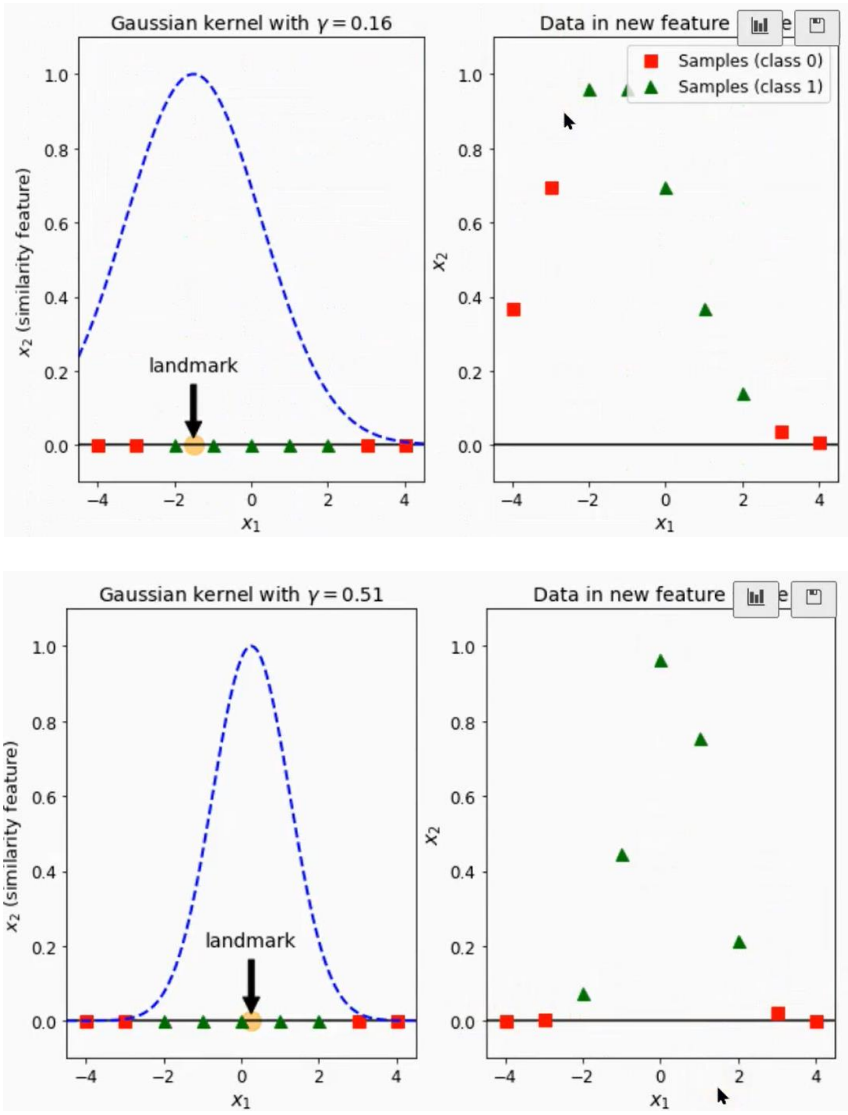
12. Given a 1D data (data with 1 feature):

1 landmark => add 1 new feature => 2D

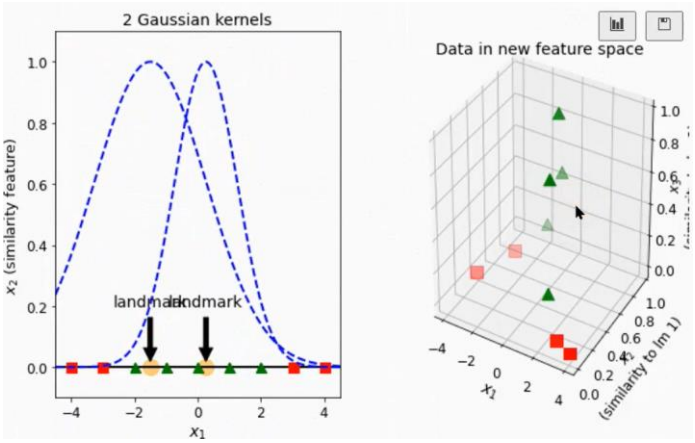
2 landmarks => add 2 new features => 3D

K landmarks => (K+1)D

13. Example: we transform 2 graph above to 2D



14. Use 2 kernels at the same time => 3D space.



3. How to select the landmarks?

15. The simplest way: each sample will turn into a landmark.

⇨ M samples turn into m landmarks. (higher dimension). Or with m sample, we add m features. We can make efficient use of multiple dimensions without having to add more dimensions to our data.

⇨ Use **Kernal trick.**

4. Implementation notes.

16. Smaller C: softer margin.

17. Smaller γ: It's going to be more complicated to learn.

5. Comparison of SVM implementation in sklearn.

Class	Time complexity	Out-of-core support*	Scaling required	Kernel trick support
LinearSVC	$O(m \times n)$	No	Yes	No
SGDClassifier*	$O(m \times n)$	Yes	Yes	No
SVC	$O(m^3 \times n)$	No	Yes	Yes

* **SGDClassifier**: Linear classifiers (SVM, logistic regression, etc.) with SGD training. Default classifier is linear SVM.
** **Out-of-core support**: not require to load all training samples into the memory for training.

CHAPTER 10: SVM REGRESSION

1. Definition

The SVM method is flexible; in addition to supporting linear and nonlinear classification, it also supports linear and nonlinear regression. The secret to using SVMs for regression rather than classification is to flip the goal: SVM Regression seeks to fit as many cases on the street as possible while reducing margin violations (i.e., instances off the street). A hyperparameter controls the street's width.

2. Math Behind SVM(A little bit)

2.1. Decision Function and Predictions

Equation 5-2. Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b \geq 0 \end{cases}$$

The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function w x + b = w x + … + w x + b.

Output : if the result is positive ,the predicted class ŷ is the positive class (1), and otherwise it is the negative class (0);

2.2. Constrained optimization problem

Think about the decision function's slope, which is the norm of the weight vector, or w. The decision boundary will be twice as far away from the spots where the decision function equals 1 if we split this slope by 2. In other words, dividing the slope by 2 will multiply the margin by 2

➔ Minimize ‖ w ‖ to get a large margin

Figure 5-13. The smaller the weight vector **w**, the larger the margin.

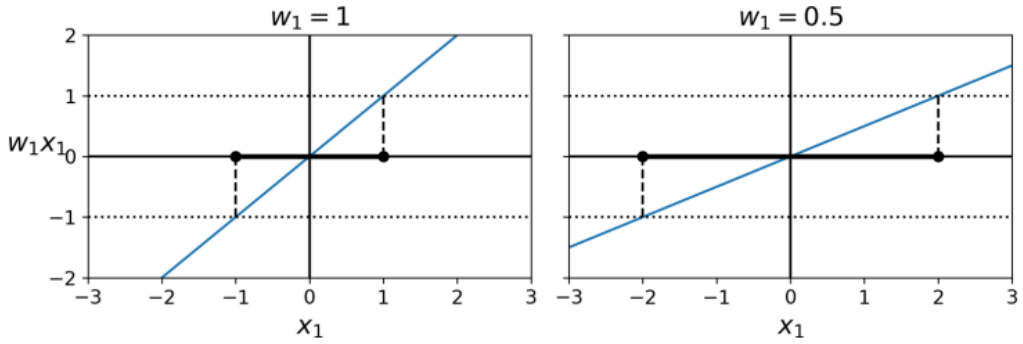


Figure 5-13. A smaller weight vector results in a larger margin

The hard margin linear SVM classifier objective :

So we want to minimize ‖ w ‖ to get a large margin. If we also want to avoid any margin violations (hard margin), then we need the decision function to be greater than 1 for all positive training instances and lower than –1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t\,(w^T\ x^{(i)} + b) \geq 1$ for all instances.

Equation 5-3. Hard margin linear SVM classifier objective

$$\begin{aligned} &\underset{\mathbf{w},b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} \\ &\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m \end{aligned}$$

NOTE

We are minimizing $\frac{1}{2}\mathbf{w}^\intercal\,\mathbf{w}$, which is equal to $\frac{1}{2}\,/\!/\,\mathbf{w}\,/\!/^2$, rather than minimizing $/\!/\,\mathbf{w}\,/\!/$. Indeed, $\frac{1}{2}\,/\!/\,\mathbf{w}\,/\!/^2$ has a nice, simple derivative (it is just **w**), while $/\!/\,\mathbf{w}\,/\!/$ is not differentiable at $\mathbf{w} = 0$. Optimization algorithms work much better on differentiable functions.

Soft margin linear SVM classifier objective :

To get the soft margin objective, we need to introduce a slack variable $\zeta^{(i)} \geq 0$ for each instance: $\zeta^{(i)}$ measures how much the i^{th} instance is allowed to violate the margin. We now have two conflicting objectives: make the slack variables as small as possible to reduce the margin violations, and make $\frac{1}{2}(\mathbf{w}^\mathbf{T})\mathbf{w}$ as small as possible to increase the margin. This is where the C hyperparameter comes in: it allows us to define the tradeoff between these two objectives.

Equation 5-4. Soft margin linear SVM classifier objective

$$\underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

2.3.The Dual Problem

The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can have the same solution as the primal problem. Luckily, the SVM problem happens to meet these conditions, so you can choose to solve the primal problem or the dual problem; both will have the same solution.

Equation 5-6. Dual form of the linear SVM objective

$$\underset{\alpha}{\text{minimize}} \ \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)\intercal}\mathbf{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

where $t^{(i)} = \begin{cases} -1 \text{ if } y^{(i)} = 0 \\ +1 \text{ if } y^{(i)} = 1 \end{cases}$ for $i = 1, 2, \ldots, m$.

Solution:

$$\hat{\mathbf{w}} = \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m}\left(t^{(i)} - \hat{\mathbf{w}}^\intercal\mathbf{x}^{(i)}\right)$$

where $\alpha^\wedge$ is the α that minimizes the dual problem.

CHAPTER 11: DECISION TREES

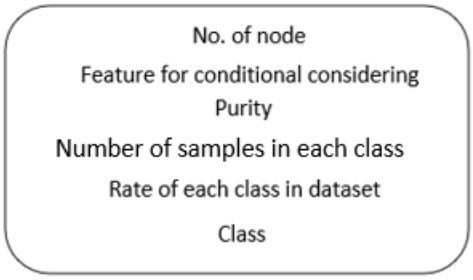
1.Definition

Like SVMs, Decision Trees are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are powerful algorithms, capable of fitting complex datasets

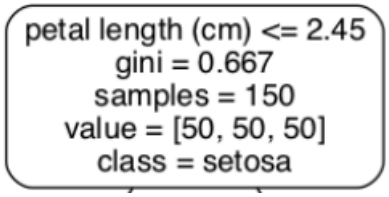
Decision Trees are also the fundamental components of Random Forests ,which are among the most powerful Machine Learning algorithms available today.

2. Visualizing a Decision Tree

**** Node's attributes:***



Example :



Making Predictions :

a) Trains a DecisionTreeClassifier on the iris dataset :

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

b) Visualize the trained Decision Tree by first using the export_graphviz() method to output a graph definition file called iris_tree.dot

```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

c) Result and Explaining :

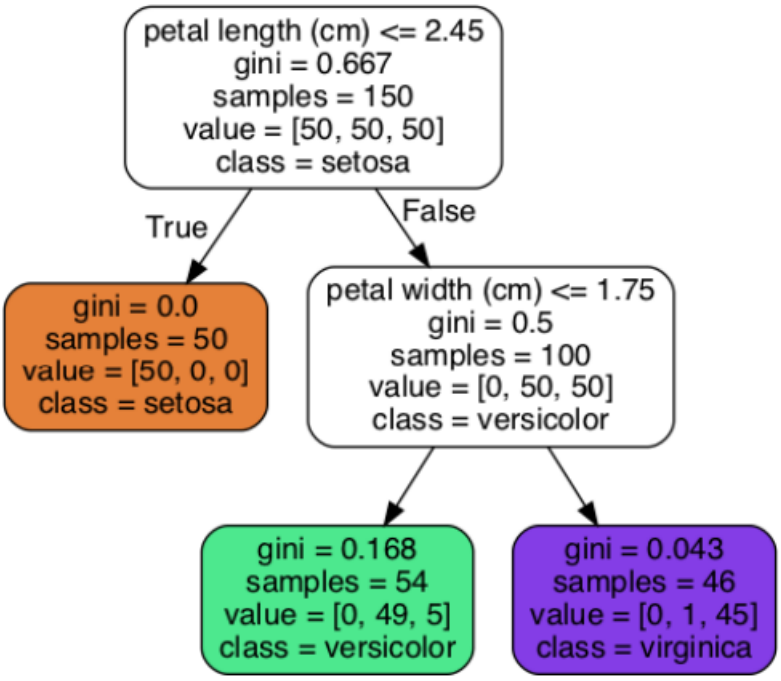


Figure 6-1. Iris Decision Tree

Picture 1

Suppose you find an iris flower and you want to classify it.

Step 1 : Starting at the root node(Node 0) with checking whether the flower's petal length is smaller than 2.45 cm

Step 2 :

If so, descend to the leftmost child node of the root (Node 2, left). The Decision Tree suggests that your flower is an Iris setosa (class = setosa) in this instance because it is a leaf node (i.e., it has no child nodes). To determine this, look at the anticipated class for that node.

Else, if the petal length is greater than 2.45 cm . The node asks another question: Is the petal width lower than 1.75 cm? You must descend to the right child node of the root (depth 1, right), which is not a leaf node. Your flower is most likely an Iris versicolor if it is (Node 3, left). Iris virginica is most likely if not (Node 4, right).

Decision Tree's decision boundaries:

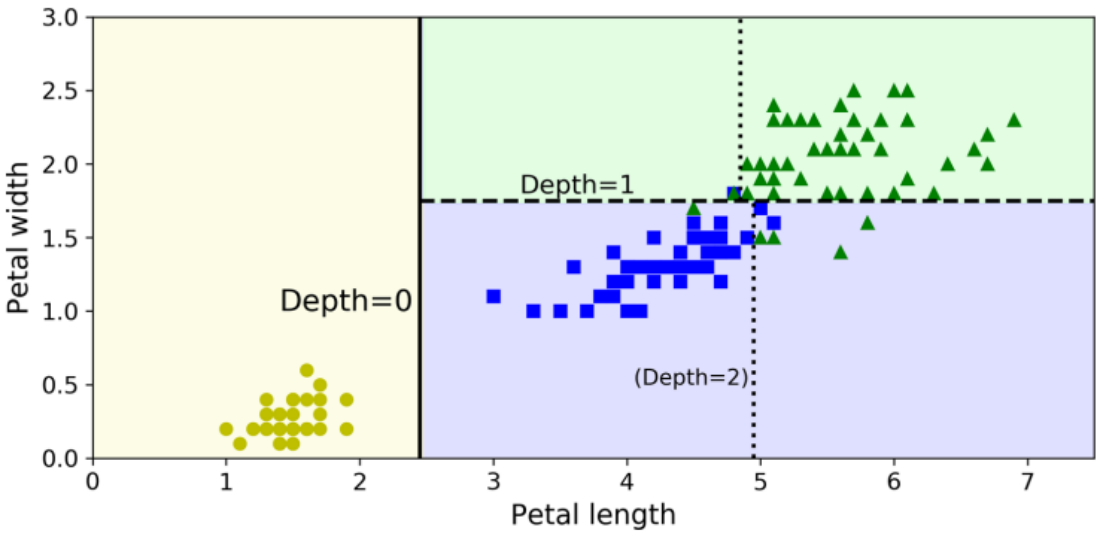


Figure 6-2. Decision Tree decision boundaries

Mechanism :

The thick vertical line denotes the root node's (depth 0) decision boundary; the petal length is 2.45 cm. The left-hand portion cannot be divided further since it contains solely Iris setosa. The right node of depth-1 separates the righthand region at petal width = 1.75 cm (shown by the dashed line) because it is impure. The Decision Tree immediately comes to an end since the max depth was set to 2. The two depth-2 nodes would each add another decision boundary if you set the max depth to 3. (represented by the dotted lines).

3. Gini – an impurity measuremen

3.1. Definition

The gini property of a node determines how impure it is: a node is "pure" (gini=0) if all of the training examples it applies to are of the same class.

3.2. Gini impurity formula

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

Picture 2

In this equation:

p is the ratio of class k instances among the training instances in the i^{th} node.

Example :

Picture 2 shows how the training algorithm computes the gini score G of the i node.

Since the Node 1 left node applies only to Iris setosa training instances, it is pure and its gini score equal to $(1- (50/50)^2-(0/50)^2-(0/50)^2) = 0$.

The depth-2 left node has a gini score equal to $1 – (0/54)^2 – (49/54)^2 – (5/54)^2 \approx 0.168$.

***The CART Training Algorithm :**

The method begins by utilizing a single feature (k) and a threshold (t) (for example, "petal length 2.45 cm") to divide the training data into two groups. How is k and t selected by it? It seeks the pair (k, t) that yields the most pure subsets (weighted by their size).

3.3. Impurity measurement

***Gini :** The gini property of a node determines how impure it is: a node is "pure" (gini=0) if all of the training examples it applies to are of the same class.

***Entropy :**

+ In thermodynamics : a measure of molecular disorder in which entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains

+ **In information theory :** it measures the average information content of a message in which entropy is zero when all messages are identical.

+ In Machine Learning : entropy is frequently used as an impurity measure: a set’s entropy is zero when it contains instances of only one class.

***Regularization for decision trees**

To restricts it freedom in training:

▪ max_depth:

▪ min_samples_split:

▪ min_samples_leaf:

▪ max_leaf_nodes:

▪ max_features:

3.4. Regression with decision trees

This tree resembles the categorization tree you constructed before quite closely. The primary distinction is that each node now predicts a value rather than a class.

Making Predictions:

a) *Using Scikit-Learn's DecisionTreeRegressor class to build a regression tree , training it on a noisy quadratic dataset with max_depth=2:*

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

b) Result and Explaining :

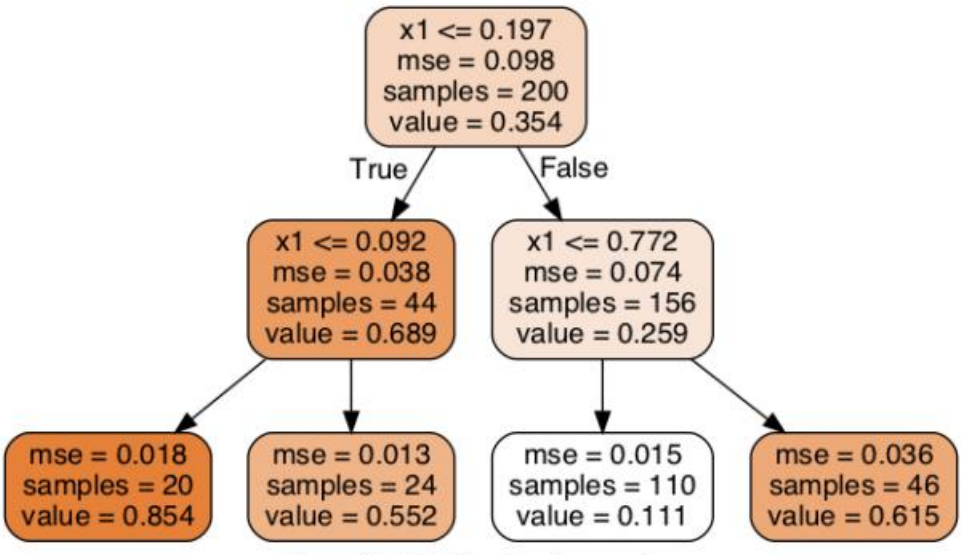


Figure 6-4. A Decision Tree for regression

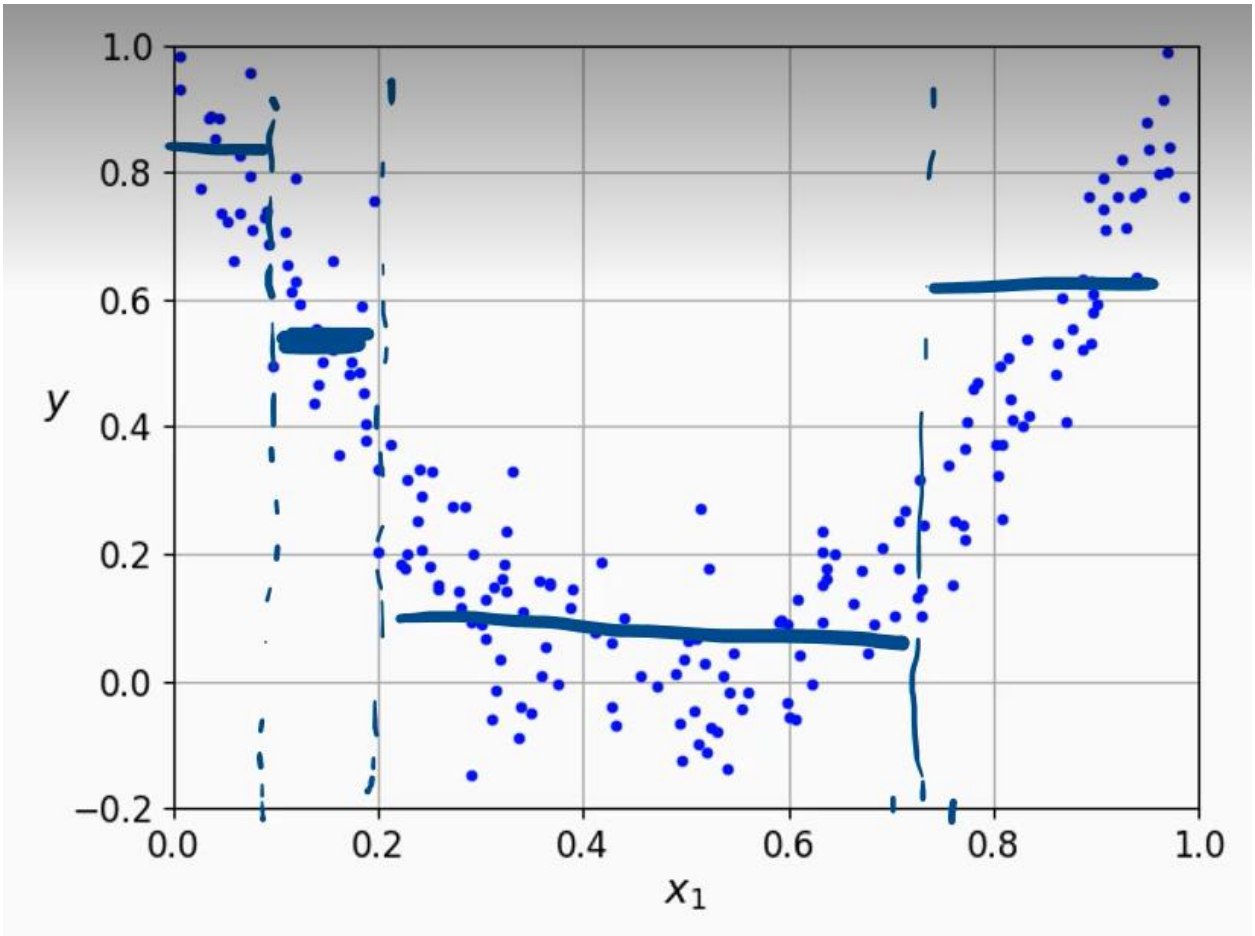
Example : suppose you want to make a prediction for a new instance with x = 0.012.

Step 1 : Starting at the root node(Node 0) with checking whether x < = 0.197 or not

Step 2 : Because it is true , descending to the leftmost child node of the root (Node 1, left) , and continue to check the second condition whether x <= 0.092 or not

Step 3 : Because it is true , descending to the leftmost child node of the root (Node 2, left), and This is the leaf node , so The Decision Tree suggests that when x = 0.012 ➔value = 0.854

* This model's predictions are represented on the graph in Picture 3*



Picture 3

Keep in mind that each region's forecast value is always the average target value of its instances. The algorithm divides each region so that the majority of training cases are as near to the anticipated value as is feasible

3.5. Limitations of decision trees

Pros and cons of decision trees:

**** Pros :*** They are simple to use, easy to comprehend and interpret, versatile, and powerful.

**** Cons :***

- Because of their powerful strength ,it is easy to get overfitting if we don't choose the right depth.

Example :

Decision Trees are prone to overfitting while handling regression problems, much like for classification tasks. You get the predictions on the left in *Picture 4* when no regularization is used (i.e., while using the default hyperparameters). These predictions are obviously overfitting the training set very badly . A far more reasonable model is produced by just setting min_samples_leaf=10, as seen on the right in *Picture 4*.

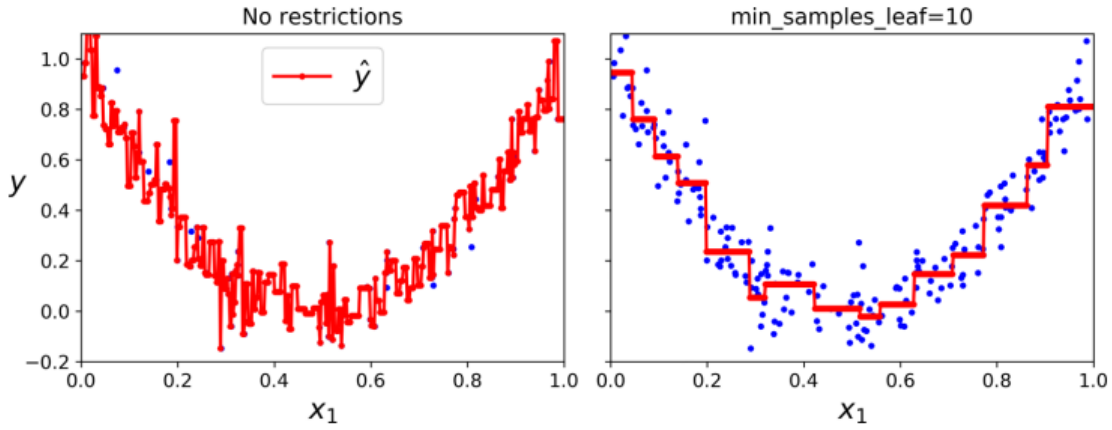


Figure 6-6. Regularizing a Decision Tree regressor

Picture 4 : Regularizing a Decision Tree regressor

- It is sensitive to data rotation with orthogonal decision boundaries (all splits are perpendicular to an axis)- the main issue is that they are very sensitive to small variations in the training data.

➔Solution : Using Principal Component Analysis which frequently yields a better orientation of the training data, is one technique to mitigate this issue.

Example :In *Picture 5*, illustrates a simple linearly separable dataset that can be readily divided by a Decision Tree on the left, but becomes needlessly complicated on the right once the dataset has been rotated by 45 degrees. While the training set was well suited by both Decision Trees, it is quite likely that the model on the right will not generalize well.

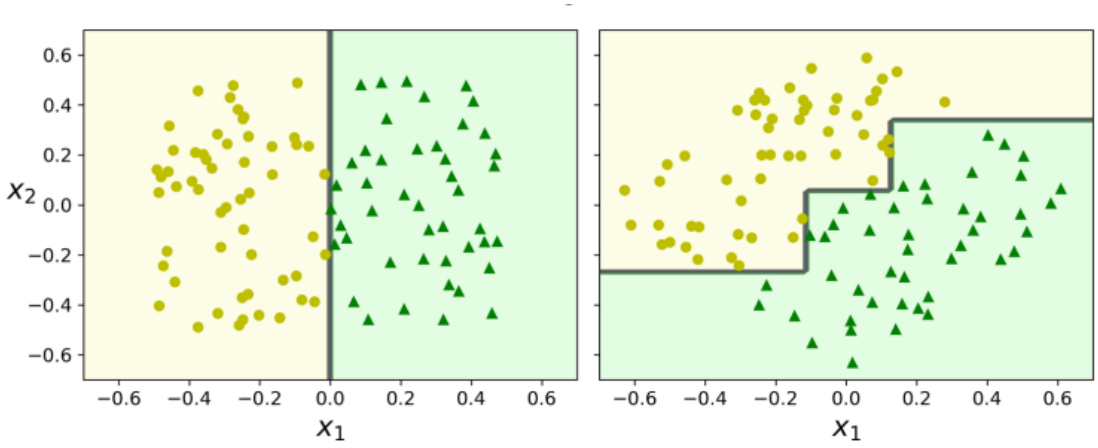


Figure 6-7. Sensitivity to training set rotation

Picture 5 . Sensitivity to training set rotation

CHAPTER 12: ENSEMBLE LEARNING

1. Introduction

1.1. Idea

The wisdom of the crowd - If you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an ensemble; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method.

1.2. Requires two conditions

1. The number of models
2. Models must be different

1.3. To get diverse classifiers(models)

1. Use different types of predictors
2. Use different training sets

2. Some Popular ensemble methods

2.1.Voting Classifiers

**** Hard voting classifiers***

Combining the predictions of each classifier and predicting the category that receives the most votes is a pretty straightforward method for developing an even better classifier.

Example :

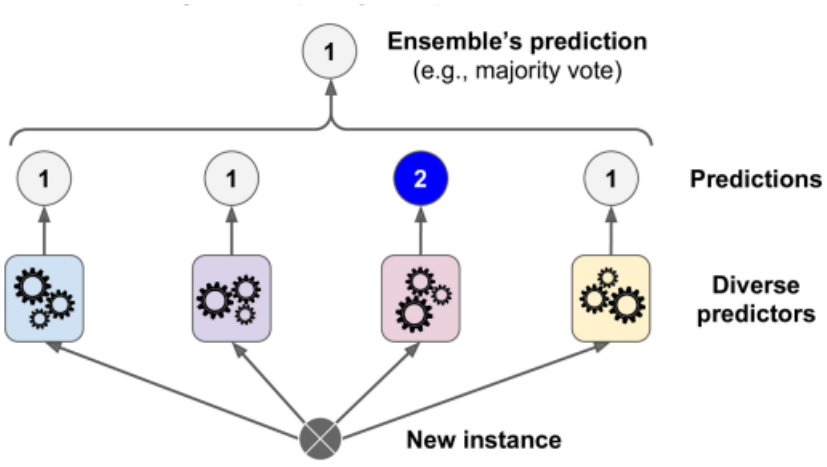


Figure 7-2. Hard voting classifier predictions

Making Predictions:

**Builds and trains a vote classifier using three different classifiers in Scikit-Learn.*

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

There you have it! The voting classifier slightly outperforms all the individual classifiers

****Soft voting classifiers :***

If all classifiers are able to estimate class probabilities , predicting the class with the highest class probability, averaged over all the individual classifiers. It often achieves higher performance than hard voting because it gives more weight to highly confident votes.

Example :

Making Predictions:

**Builds and trains a vote classifier using three different classifiers in Scikit-Learn.*

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

All you need to do is replace voting="hard" with voting="soft" and ensure that all classifiers can estimate class probabilities.

Let's look at each classifier's accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
```

Soft voter computes average class probabilities and decides

```
LogisticRegression , class proba:  [[0.29 0.71]]
RandomForestClassifier , class proba:  [[0.93 0.07]]
SVC , class proba:  [[0.79 0.21]]
VotingClassifier , class proba:  [[0.67 0.33]]  ==> predicted class  [0]
```

2.2. BAGGING METHODS

2.2.1. BAGGING METHODS

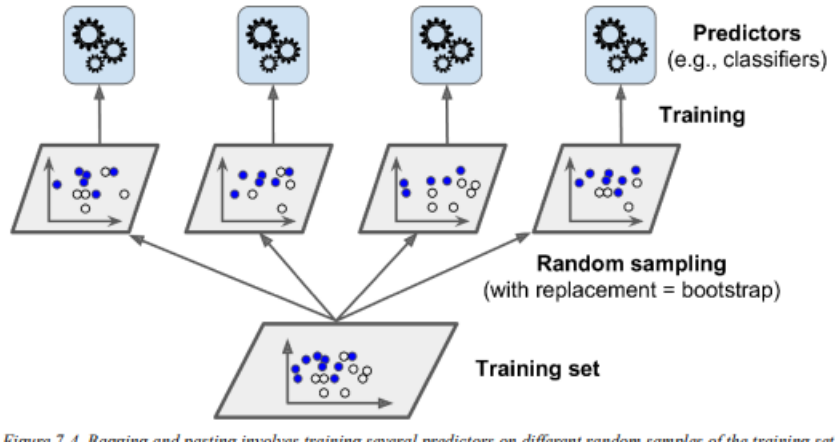
**** Definition :***

Using the same training algorithm for every predictor and train them on different random subsets of the training set and sampling is performed with replacement. bagging allows training instances to be sampled several times for the same predictor performed with replacement , this method is called bootstrapping.

When sampling is performed without replacement, it is called Non-bootstrapping

**** Main steps :***

1. Create different training sets
2. Train predictors and aggregate them to predict



Picture 4.sampling and training process

When all predictors have been trained, the ensemble may forecast a new instance by just averaging all of the predictors' predictions. *The statistical mode* (i.e., the most common prediction, exactly as a hard voting classifier) for classification or the average for regression is often the aggregation function. Although aggregation lowers both bias and variance, each individual predictor has a greater bias than if it were trained on the first training set. In general, the ensemble has a smaller variance but a similar bias as a single predictor trained on the first training set.

As you can see in Picture 4, predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons Bootstrapping and Non-bootstrapping are such popular method: they scale very well.

Example :

Scikit-Learn offers a simple API for both bagging and pasting with the BaggingClassifier class (or BaggingRegressor for regression). Trains an ensemble of 500 Decision Tree classifiers: each is trained on 100 training instances randomly sampled from the training set with replacement

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Result :

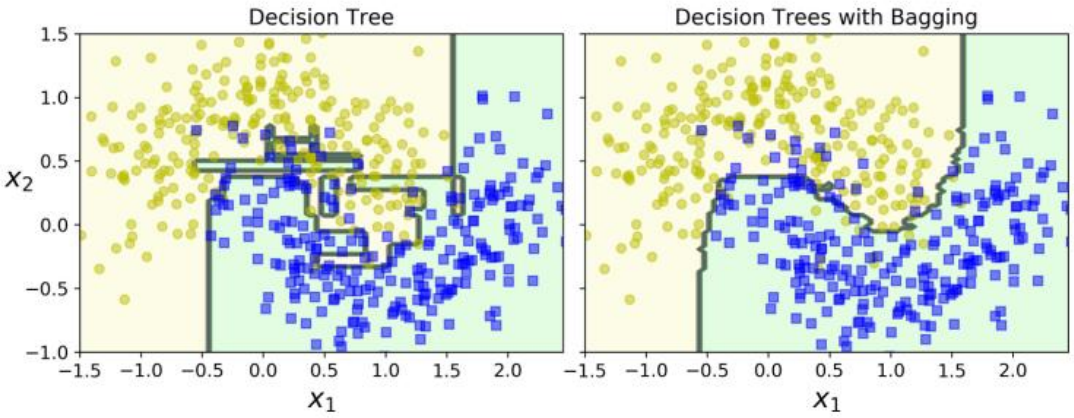


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

As you can see, the ensemble has a comparable bias but a smaller variance (it makes nearly the same amount of errors on the training set, but the decision boundary is less irregular), therefore its predictions will likely generalize considerably better than those of the single Decision Tree.

2.2.2. Random Forests

A random forest is just an ensemble of decision trees trained with bagging method.

The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features. The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. The following code uses all available CPU cores to train a Random Forest classifier with 500 trees (each limited to maximum 16 nodes):

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

When building trees, the Random Forest algorithm adds additional randomization; instead of looking for the best feature when splitting a node , it looks for the best feature among a random collection of features. Greater tree diversity is produced by the algorithm, which (again) trades a higher bias for a lower variance and generally produces a better model. Comparable to the previous RandomForestClassifier is the BaggingClassifier

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

2.2.3. Extra-Trees

Only a random subset of the characteristics are taken into account for splitting while building a tree in a Random Forest (as discussed earlier). Instead of trying to find the optimal thresholds, it is feasible to make trees even more random by applying random thresholds for each feature (like regular Decision Trees do).

An Extremely Randomized Trees ensemble is a forest of such wildly erratic trees (or Extra-Trees for short). Once more, this strategy exchanges greater bias for less variance. Because one of the most time-consuming jobs of developing a tree is determining the optimal threshold for each feature at each node, Extra-Trees are also significantly faster to train than conventional Random Forests.

2.3. Boosting Methods

Boosting, which was initially known as hypothesis boosting, refers to any Ensemble approach that may turn a number of weak learners into a powerful learner. The main principle behind most boosting techniques is to train predictors in a sequential manner while attempting to correct each one's predecessor. The following code uses all available CPU cores to train a Random Forest classifier with 500 trees (each limited to maximum 16 nodes):

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

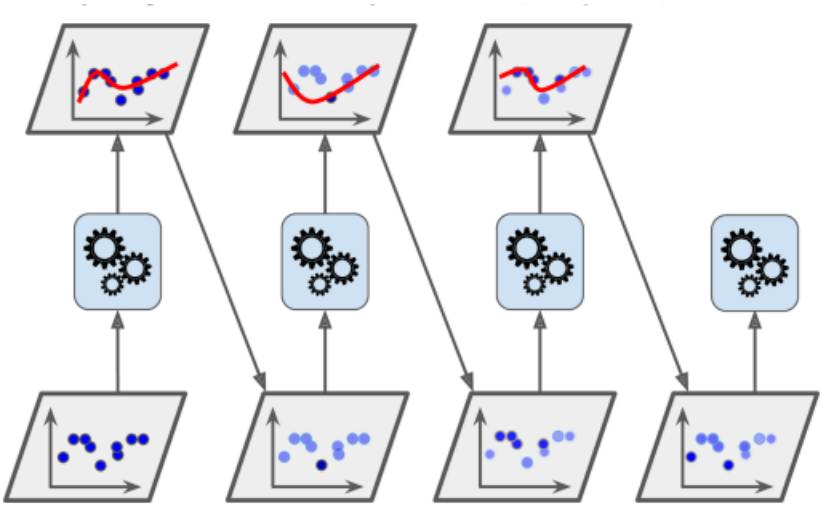
```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

2.3.1.AdaBoost

Paying a little more attention to the training examples that the predecessor underfitted is one approach for a new predictor to rectify its predecessor. As a result, new predictors increasingly concentrate on the challenging scenarios. This is the method AdaBoost employs.

When training an AdaBoost classifier, the algorithm first trains a base classifier (such as a Decision Tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on

Example :



Step 1 : A first predictor is trained, and its weighted error rate r is computed on the training set

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

using equation where $y^{(i)}_j$ is the j^{th} predictor's prediction for the i^{th} instance.

Step 2 : the AdaBoost algorithm updates the instance weights, using Equation

$$\text{for } i = 1, 2, \cdots, m$$
$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

, which boosts the weights of the misclassified instances.

Step 3 : the instance weights are updated, a new predictor is trained using the updated weights, and the entire process is repeated (the weight of the new predictor is calculated, the instance weights are updated, and so on). When the target number of predictors is obtained or a perfect predictor is discovered, the algorithm terminates

2.3.2. Gradient Boosting

**** Definition :***

Gradient Boosting is a different boosting method that is particularly well-liked. Gradient Boosting operates similarly to AdaBoost by adding predictors to an ensemble one at a time, with each one correcting the one before it. But unlike AdaBoost, this strategy seeks to adapt the new predictor to the residual errors created by the prior predictor rather than adjusting the instance weights at each iteration.

****Residual Errors***

A measurement used to evaluate how well a linear regression model fits the data is the residual standard deviation (also known as the residual standard error).

Example :

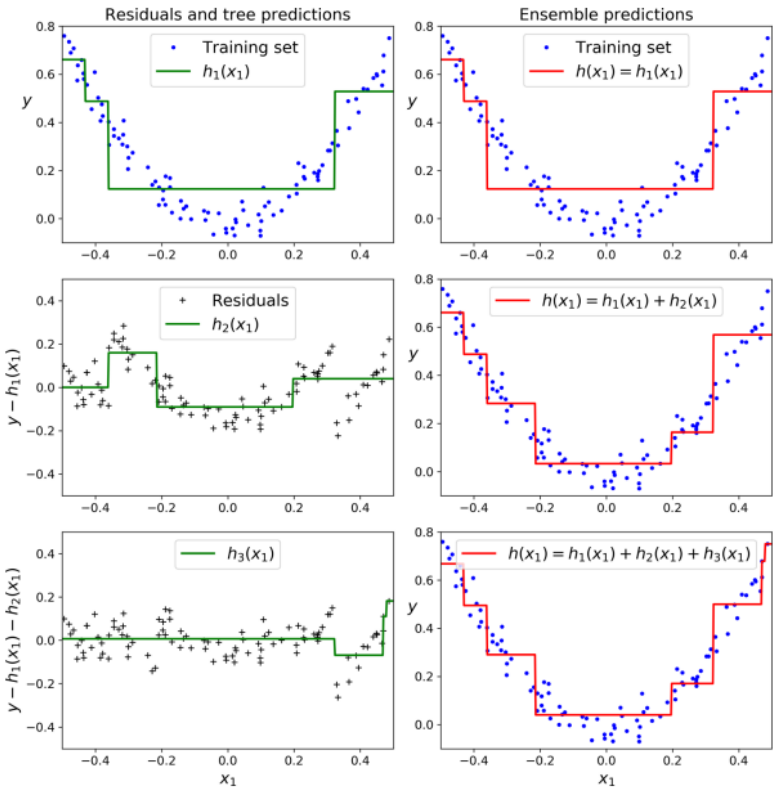


Figure 7-9. In this depiction of Gradient Boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions

Step 1 : Fit base predictors to the training set

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

→ Represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. Because there is just one tree in the ensemble, all of its predictions coincide with those of the initial tree.

Step 2 : we'll train a second Predictor on the residual errors made by the first predictor

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

→ On the first tree's leftover mistakes, a new tree is trained. The forecasts of the ensemble, as seen on the right, are equal to the combined projections of the first two trees.

Step 3 : we train a third regressor on the residual errors made by the second predictor

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

→ another tree is trained on the residual errors of the second tree.

Step 4 : Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

→ As more trees are included in the ensemble, the predictions become steadily more accurate.

CHAPTER 13: APPLICATION – FINAL PROJECT

Part 0: Import the librabries

```
#Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# import warnings
# warnings.filterwarnings('ignore')
```

Part 1: Problem understanding - look at the big picture

- End goal : "Item_Outlet_Sales"
- Type of MLs Approach : Regression
- Feature :

  +) Feature of Item: 'Item_Identifier', 'Item_Weight', 'Item_Fat_Content', 'Item_Visibility', 'Item_Type', 'Item_MRP',

  +) Feature of Outlet : 'Outlet_Identifier', 'Outlet_Establishment_Year', 'Outlet_Size', 'Outlet_Location_Type', 'Outlet_Type'

Part 2: Get and understand the data

```
train=pd.read_csv('Train.csv')
test=pd.read_csv('Test.csv')
```

Variable	Description	Relation to Hypothesis
Item_Identifier	Unique product ID	ID Variable
Item_Weight	Weight of product	Not considered in hypothesis
Item_Fat_Content	Whether the product is low fat or not	Linked to 'Utility' hypothesis. Low fat items are generally used more than others
Item_Visibility	The % of total display area of all products in a store allocated to the particular product	Linked to 'Display Area' hypothesis.
Item_Type	The category to which the product belongs	More inferences about 'Utility' can be derived from this.
Item_MRP	Maximum Retail Price (list price) of the product	Not considered in hypothesis
Outlet_Identifier	Unique store ID	ID Variable
Outlet_Establishment_Year	The year in which store was established	Not considered in hypothesis
Outlet_Size	The size of the store in terms of ground area covered	Linked to 'Store Capacity' hypothesis
Outlet_Location_Type	The type of city in which the store is located	Linked to 'City Type' hypothesis.
Outlet_Type	Whether the outlet is just a grocery store or some sort of supermarket	Linked to 'Store Capacity' hypothesis again.
Item_Outlet_Sales	Sales of the product in the particular store. This is the outcome variable to be predicted.	Outcome variable

Part 3. Discover the data to gain insights

3.1. Quick view of the data

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	1999	Medium	Tier 1	Supermarket Type1	3735.1380
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009	Medium	Tier 3	Supermarket Type2	443.4228
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	1999	Medium	Tier 1	Supermarket Type1	2097.2700
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	OUT010	1998	NaN	Tier 3	Grocery Store	732.3800
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	OUT013	1987	High	Tier 3	Supermarket Type1	994.7052

```
print(train.shape)
print(test.shape)
train["source"] = "train"
test["source"] = "test"
test["Item_Outlet_Sales"] = 0.0
df = pd.concat([train, test], sort = False, ignore_index = True)
print(df.shape)
display(df.head())
```

```
(8523, 12)
(5681, 11)
(14204, 13)
```

- Train set: 8532 samples, 12 features.
- Test set: 5681 samples, 11 features.
- We will combine train and test data for better Analysis: 14204 samples, 13 features.
- Data Describe: Data of Test does not have target 'Item_Outlet_Sales' ==> It should combine train and test into one to make the analysis and transformation. After that, we seperate back to train and test.

3.2. Check Missing Values

- Item weight column has about 2439 missing values and outlet size column has 4016 NaN values.

- Datset has 8 rows with object type data out of which we have created one and one is an identifier column. others we will have to encode them.

- There is skewness present in the item visibility which need to be handled.

```
df.isnull().sum()
✓ 0.1s
Item_Identifier                 0
Item_Weight                  2439
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  4016
Outlet_Location_Type            0
Outlet_Type                     0
Item_Outlet_Sales               0
source                          0
dtype: int64
```

```
df.dtypes
✓ 0.1s
Item_Identifier                object
Item_Weight                   float64
Item_Fat_Content               object
Item_Visibility               float64
Item_Type                      object
Item_MRP                      float64
Outlet_Identifier              object
Outlet_Establishment_Year       int64
Outlet_Size                    object
Outlet_Location_Type           object
Outlet_Type                    object
Item_Outlet_Sales             float64
source                         object
dtype: object
```

```
df.skew()
✓ 0.7s
C:\Users\hung\AppData\Local\Temp\ipyke
a future version, it will default to F
of numeric_only to silence this warnin
  df.skew()

Item_Weight                  0.101309
Item_Visibility              1.195175
Item_MRP                     0.130728
Outlet_Establishment_Year   -0.396465
Item_Outlet_Sales            1.544684
dtype: float64
```

- There are null values present in item weight as count less than total rows, mean is less than median in item MRP and outlet establishment year, rest of the features have median greater than median. Variance is almost zero in item visibility column and very high in item MRP. There are some outliers present as difference between min, max and interquartile range is unequal. Minimum value of visibility is zero that can not be possible so we will treat it as a null value.

```
display(df.describe())
display(df.describe(include = object))
```

	Item_Weight	Item_Visibility	Item_MRP	Outlet_Establishment_Year	Item_Outlet_Sales
count	11765.000000	14204.000000	14204.000000	14204.000000	14204.000000
mean	12.792854	0.065953	141.004977	1997.830681	1308.865489
std	4.652502	0.051459	62.086938	8.371664	1699.791423
min	4.555000	0.000000	31.290000	1985.000000	0.000000
25%	8.710000	0.027036	94.012000	1987.000000	0.000000
50%	12.600000	0.054021	142.247000	1999.000000	559.272000
75%	16.750000	0.094037	185.855600	2004.000000	2163.184200
max	21.350000	0.328391	266.888400	2009.000000	13086.964800

	Item_Identifier	Item_Fat_Content	Item_Type	Outlet_Identifier	Outlet_Size	Outlet_Location_Type	Outlet_Type	source
count	14204	14204	14204	14204	10188	14204	14204	14204
unique	1559	5	16	10	3	3	4	2
top	FDU15	Low Fat	Fruits and Vegetables	OUT027	Medium	Tier 3	Supermarket Type1	train
freq	10	8485	2013	1559	4655	5583	9294	8523

- **Missing Values:** Look at count in describe : Missing Value occur at features: Item_Weight(17.17%) and Outlet_Size(28.27%)**.** Investigate some unreasonable value at feature : Item_Visibility with value 0 (6.18%)

3.3. Check Outlier(Noise)

- Code : analyze the each number feature

```
df1 = df[df["Item_Visibility"] > 0]

Stats = df1.describe()
for col in Stats.columns:
  meanV, stdV = Stats.loc["mean", col], Stats.loc["std", col]
  minV, maxV = Stats.loc["min", col], Stats.loc["max", col]
  q1, q2, q3 = Stats.loc["25%", col], Stats.loc["50%", col], Stats.loc["75%", col]
  iqr = q3 - q1
  ConfRange1 = [round(meanV - 3 * stdV,2), round(meanV + 3 * stdV,2)]
  ConfRange2 = [round(q1 - 1.5 * iqr,2), round(q3 + 1.5 * iqr,2)]
  print(f"{col} : \n\tMin is {minV} and Max is {maxV} \n\tConfident Range 1 is {ConfRange1} and Range 2 is {ConfRange2}")
```

- Result :

```
Item_Weight :
        Min is 4.555 and Max is 21.35
        Confident Range 1 is [-1.15, 26.75] and Range 2 is [-3.35, 28.81]
Item_Visibility :
        Min is 0.003574698 and Max is 0.328390948
        Confident Range 1 is [-0.08, 0.22] and Range 2 is [-0.07, 0.2]
Item_MRP :
        Min is 31.29 and Max is 266.8884
        Confident Range 1 is [-44.93, 327.14] and Range 2 is [-43.43, 323.43]
Outlet_Establishment_Year :
        Min is 1985.0 and Max is 2009.0
        Confident Range 1 is [1972.7, 2022.96] and Range 2 is [1961.5, 2029.5]
Item_Outlet_Sales :
        Min is 0.0 and Max is 13086.9648
        Confident Range 1 is [-3785.48, 6400.42] and Range 2 is [-3245.77, 5409.62]
```

➔ Conclusion : Feature having outlier values is Item_Visibility because the Max value of this feature is out of both Confident Range 1 and Confident range 2 .

3.4. Check Inconsistent

```
CatFeatures = [col for col in df.columns if df[col].dtypes in ["object"]]
NumFeatures = [col for col in df.columns if df[col].dtypes not in ["object"]]
print("Numeric Features : ", NumFeatures)
print("Categorical Features : ", CatFeatures)

for col in CatFeatures:
  print(f"\n{col} : \n{df[col].value_counts()}")
```

```
Item_Identifier :
FDU15    10
FDS25    10
FDA38    10
FDW03    10
FDJ10    10
        ..
FDR51     7
FDM52     7
DRN11     7
FDH58     7
NCW54     7
Name: Item_Identifier, Length: 1559, dtype: int64

Item_Fat_Content :
Low Fat    8485
Regular    4824
LF          522
reg         195
low fat     178
Name: Item_Fat_Content, dtype: int64

Item_Type :
Fruits and Vegetables    2013
Snack Foods              1989
Household                1548
Frozen Foods             1426
Dairy                    1136
Baking Goods             1086
Canned                   1084
Health and Hygiene        858
Meat                      736
Soft Drinks               726
Breads                    416
Hard Drinks               362
Others                    280
Starchy Foods             269
Breakfast                 186
Seafood                    89
Name: Item_Type, dtype: int64
```

➔ Inconsistent: Low Fat & LF & low fat is the same meaning, Regular & reg is the same meaning.

3.5. Check Imbalanced

```
# Imbalanced ?
df["Item_Outlet_Sales"].skew()
```

```
1.5446838706795227
```

➔Conclusion : It now skew so much because the skewness is less than 4

```
# Skewness ?
df.skew()
```

```
Item_Weight                0.101309
Item_Visibility            1.195175
Item_MRP                   0.130728
Outlet_Establishment_Year -0.396465
Item_Outlet_Sales          1.544684
dtype: float64
```

➔Conclusion : There is no skewed feature because all their skewness are less than 4

3.6. EDA with Visualization on Univariate, BiVariate and MultiVariate

Analysis

```
print('Categorical Features: ',CatFeatures)
print('\nNumeric Features: ',NumFeatures)
cat = list(set(CatFeatures) - set("source"))
print('\nCategorical Features (without source):',cat)
```

```
['Item_Identifier', 'Item_Fat_Content', 'Item_Type', 'Outlet_Identifier', 'Outlet_Size', 'Outlet_Location_Type', 'Outlet_Type', 'source']
['Item_Weight', 'Item_Visibility', 'Item_MRP', 'Outlet_Establishment_Year', 'Item_Outlet_Sales']
['Outlet_Size', 'Outlet_Identifier', 'source', 'Item_Type', 'Outlet_Type', 'Item_Identifier', 'Item_Fat_Content', 'Outlet_Location_Type']
```

- Categorical Features : Bar Chart (Comparation), Pie (Percentage Measurements) , [TreeMap, ClusterMap]

- Numeric Features : Histogram (Distribution Shape), Boxplot (Range and Outlier) , [TreeMap, ClusterMap]

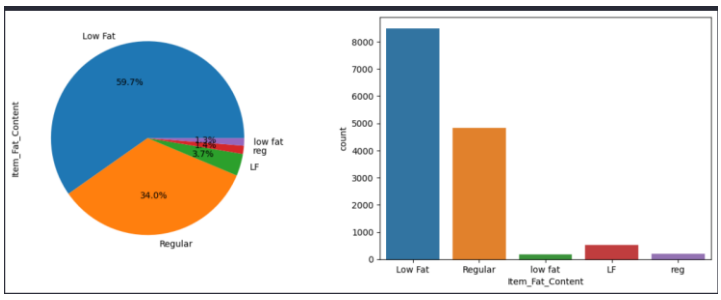
3.7. Exploring Data Analysis

3.7.1. Univariate Analysis

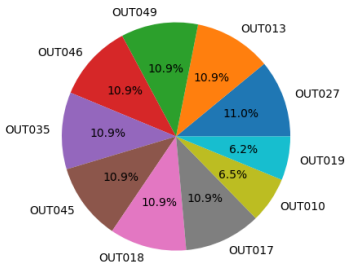
a) Categorical Features

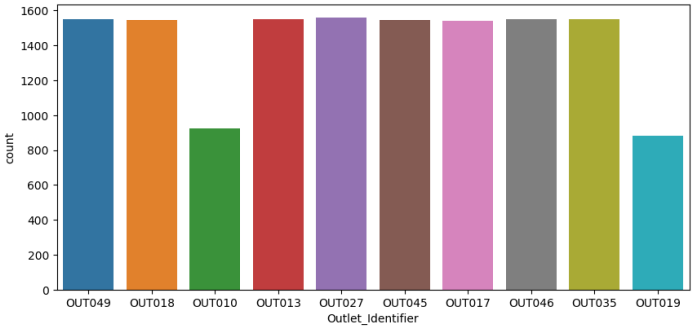
Pie (Percentage Measurements) and Bar Chart (Comparation)

- *"Item_fat_content" features :*

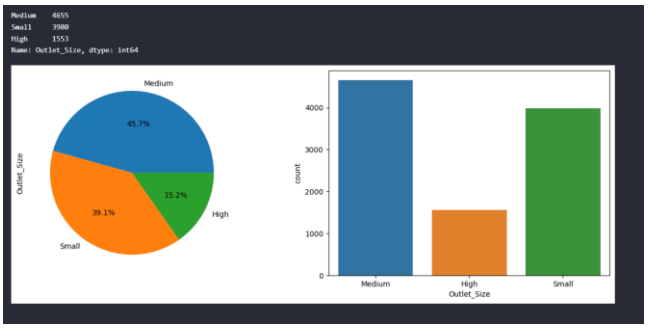


- *"Outlet_Identifier" feature :*

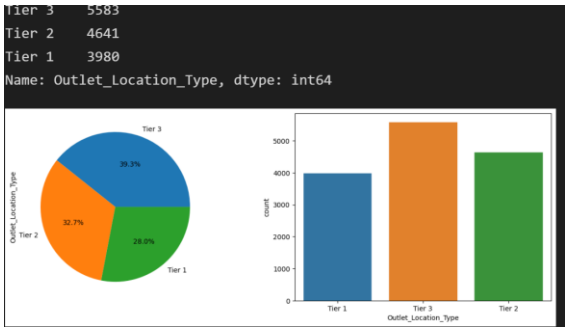

OUT049
OUT013
OUT046
10.9%
10.9%
10.9%
OUT027
11.0%
OUT035
10.9%
6.2%
OUT019
10.9%
6.5%
OUT045
10.9%
10.9%
OUT010
OUT018
OUT017


1600
1400
1200
1000
800
600
400
200
0
count
OUT049
OUT018
OUT010
OUT013
OUT027
OUT045
OUT017
OUT046
OUT035
OUT019
Outlet_Identifier

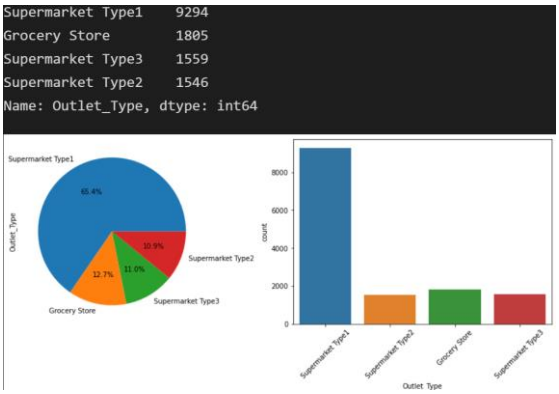
- *"Outlet_size" :* There are few outlets with high size, most of the outlets are medium size.



- *"Outlet_Location_Type":* Most of the stores are located in tier 3 cities

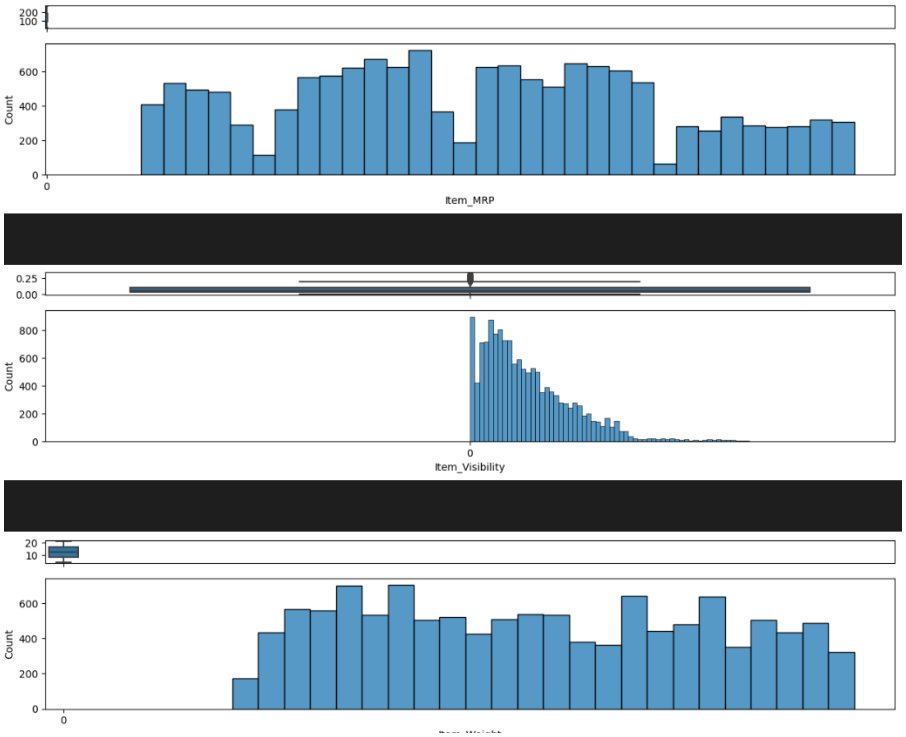


- *"Outlet_Type":* 65% of the outlets are of supermarket type 1



b) Numeric Features :

List all the features except "Outlet_Establishment_Year" and "Item_Outlet_Sales" Histogram (Distribution Shape) and Boxplot (Range and Outlier) virtualization: Virtualize and show boxplot and Histogram for each feature in the continue array that we filtered out in the previous step :

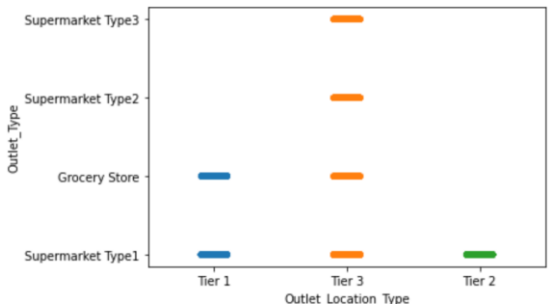


➔Conclusion : Only Item Visibility has large no. of outliers and it is skewed to the right while others almost follow the gaussian distribution.

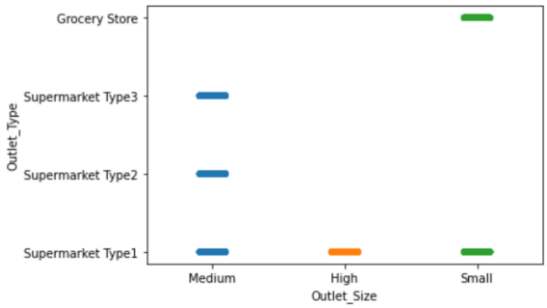
3.7.2. Bivariate Analysis (Scatter plot b/w 2 features)

a) Categorical vs Categorical (stripplot)

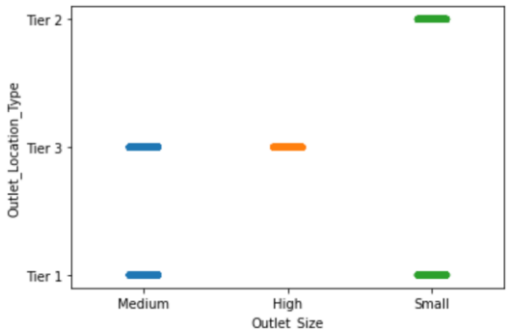
- *Outlet_Location_Type vs Outlet_Type :* Tier 2 cities have only Supermarket type 1 whereas Tier 1 cities have only supermarket 1 and grocery stores



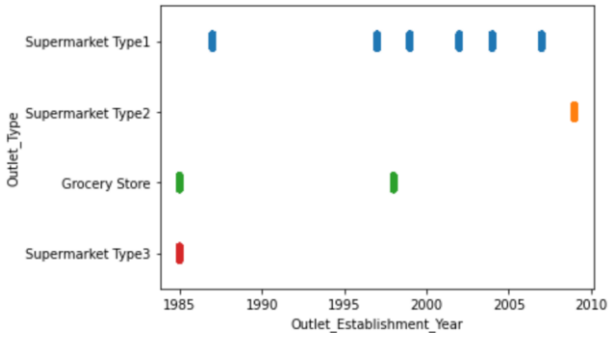
- *Outlet_Size and Outlet_Type:* Supermarket type 1 are of all sizes whereas grocery stores are only small and Supermarket type 2, Supermarket type 3 are of medium size only.



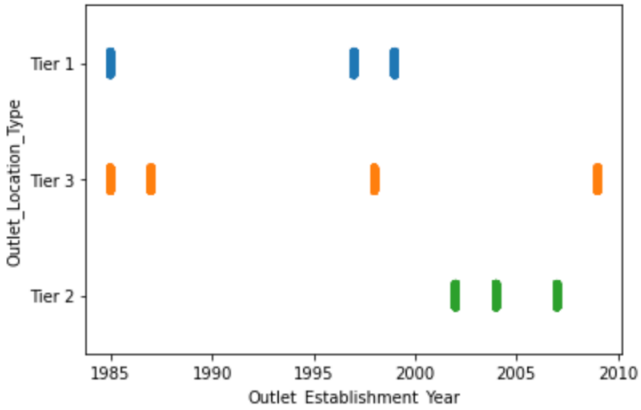
- *Outlet_Size and Outlet_Location_Type: Tier 2 citis have only small outlet size and high outlet size is only found in tier 3 cities*



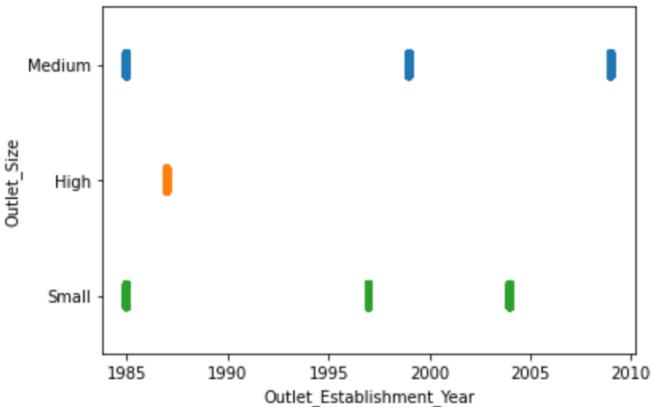
- *Outlet_Establishment_Year and Outlet_Type:* Supermarket type 2 was build much later while grocery stores and supermarket are the oldest outlet type.



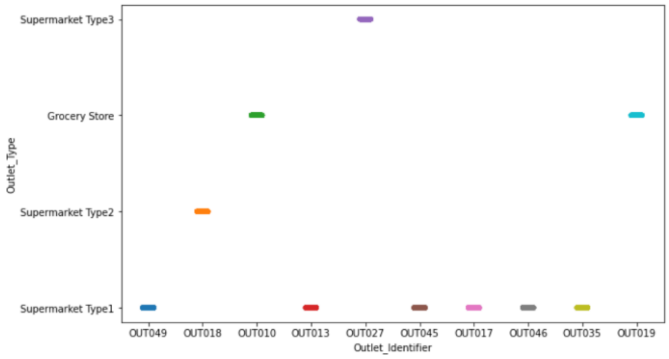
- *Outlet_Establishment_Year and Outlet_Location_Type:* In Tier1 and tier2 cities outlets were established in 1985 whereas tier2 got outlets after 2000



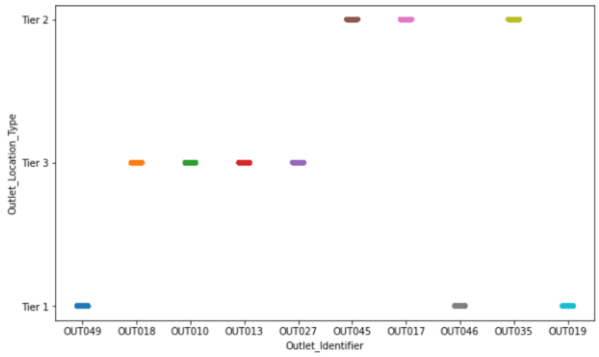
- *Outlet_Establishment_Year and Outlet_Size: After 1990 no outlet of high size was established.*



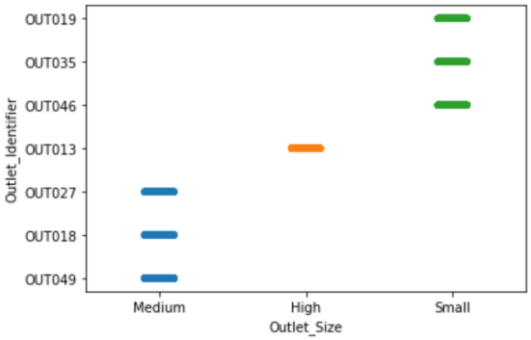
- *Outlet_Identifier and Outlet_Type:* There is only one outlet identifier for supermarket 2 and 3 while most of the outlet identifiers belong to supermarket1



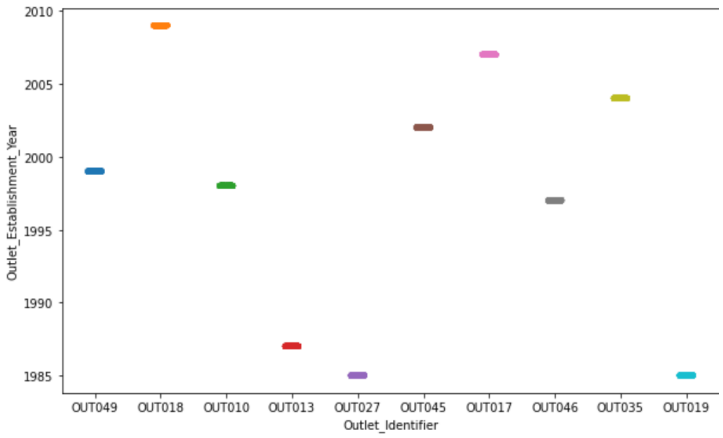
- *Outlet_Identifier and Outlet_Location_Type:* Tier 3 has the most different types of outlets, but they are almost balanced.



- *Outlet_Size and Outlet_Identifier: There is only Outlet13 with high outlet size while medium and small of outlets each have 3 outlet identifiers.*

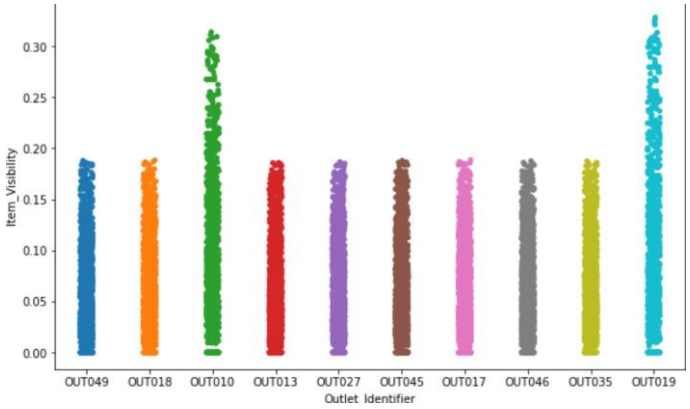


- *Outlet_Identifier and Outlet_Establishment_Year:* Outlet 27 and 19 are the oldest outlet identifier and outlet 18 is the newest, outlet no. does not hold any order towards year of establishment.

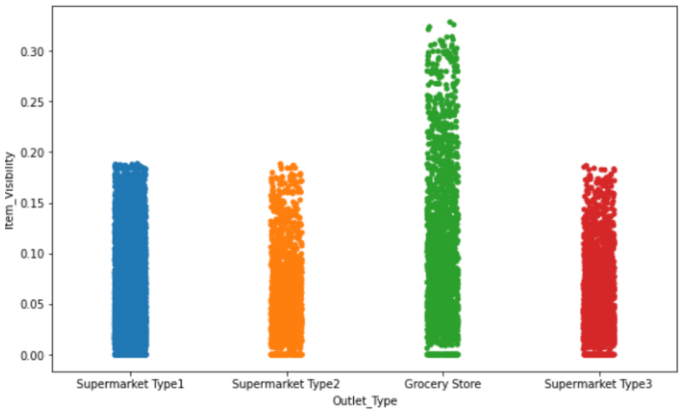


b) Categorical vs Numeric :

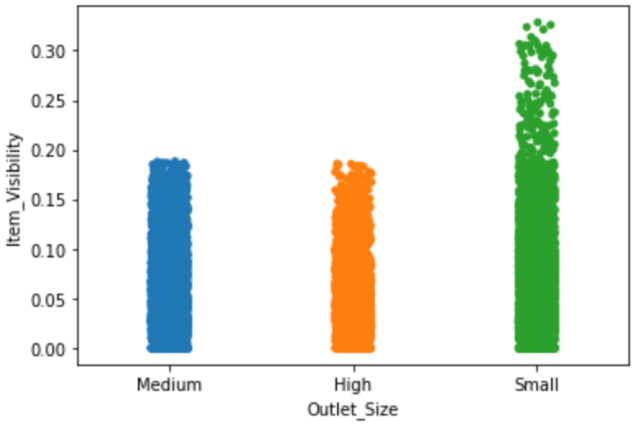
- *Outlet_Identifier vs Item_Visibility : Outlet 10 and 19 have give the highest visibilty to products. Other outlets provide almost equal visibility.*



- *Outlet_Type vs Item_Visibility :* Products are most visible in grocery stores rather than any other super markets

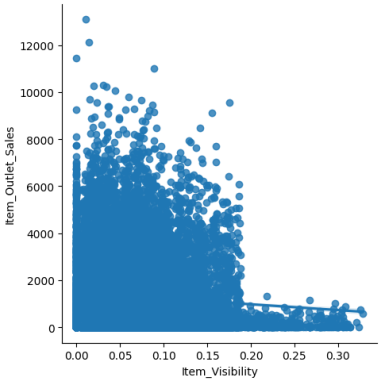


- *Outlet_Size and Outlet_SizeItem_Visibility:* Smallest Stores provide the most visibility to products

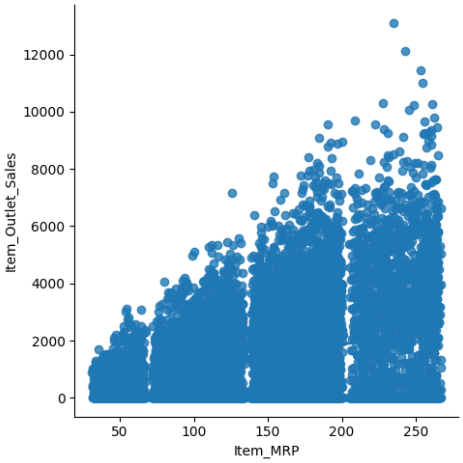


c) Numeric vs Numeric:

- *Item_Visibility vs Item_Outlet_Sales*: Item outlet sales sharply decreases for the most visible items



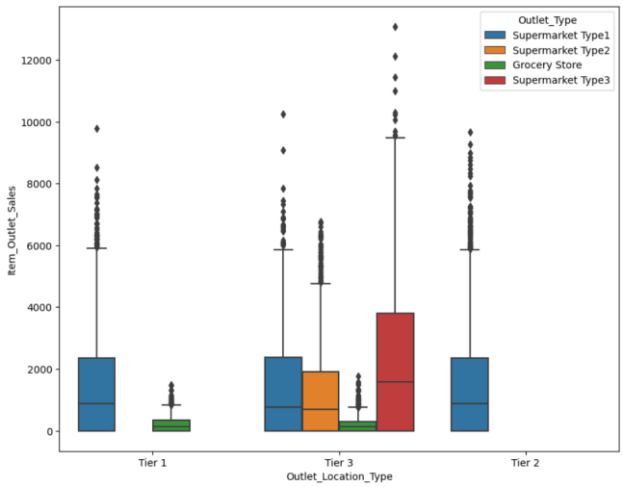
- *Item_MRP vs Item_Outlet_Sales :* As the mrp of an item increases item outlet sales also increases



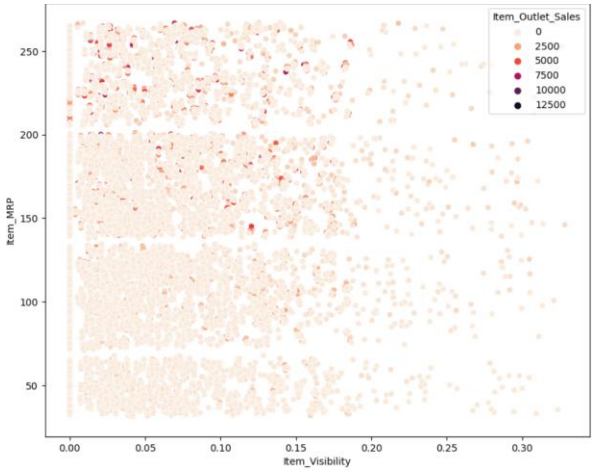
3.7.3. Multivariate Analysis

- *Outlet_Size vs Item_Outlet_Sales vs Outlet_Type :* It can be seen that medium size outlets sell the most and also they have the most type of outlets hence the sales increases even more

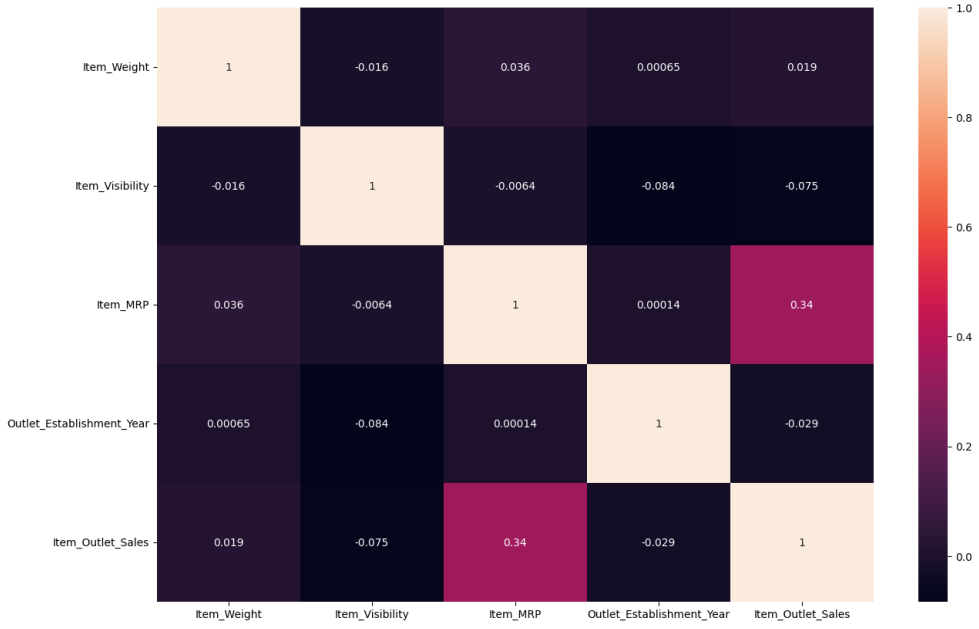
```
<AxesSubplot: xlabel='Outlet_Location_Type',
ylabel='Item_Outlet_Sales'>
```



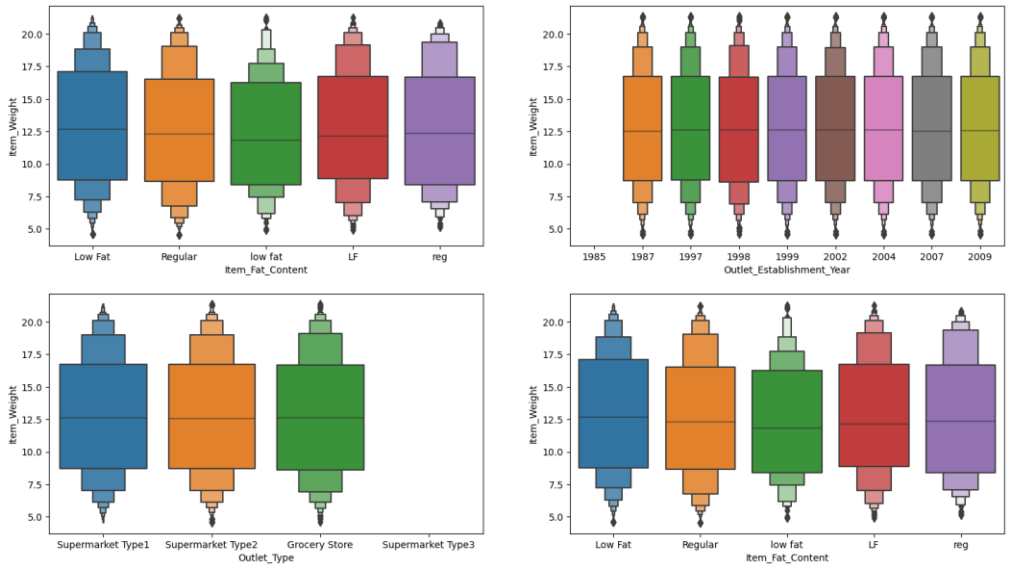
- *Item_Visibility vs Item_MRP:* Item_Outlet_Sales increases with low item visibility and high item price.



- *All of features of Numerical features set ([TreeMap, ClusterMap]):* MRP has the strongest positive correlation with Sales, while cisibility and establishment show a little negative correlation. There is not much correlation between the independent features.



- *Item_Fat_Content vs Outlet_Establishment_Year vs Outlet_Type vs Item_Fat_Content (Boxenplot):*



3.8. Compute correlations b/w features

***Correlation Between Numeric with Numeric**

	Item_Weight	Item_Visibility	Item_MRP	Outlet_Establishment_Year	Item_Outlet_Sales
Item_Weight	1.000000	-0.018540	0.036236	0.000645	0.019447
Item_Visibility	-0.018540	1.000000	-0.009230	-0.091455	-0.080968
Item_MRP	0.036236	-0.009230	1.000000	0.000141	0.342781
Outlet_Establishment_Year	0.000645	-0.091455	0.000141	1.000000	-0.029489
Item_Outlet_Sales	0.019447	-0.080968	0.342781	-0.029489	1.000000

***Correlation Between Categorical with Categorical**

	Item_Identifier	Item_Fat_Content	Item_Type	Outlet_Identifier	Outlet_Size	Outlet_Location_Type	Outlet_Type	source
Item_Identifier	1.000000	0.574985	1.000000	0.102063	0.073799	0.082409	0.120311	0.328087
Item_Fat_Content	0.574985	1.000000	0.215021	0.024022	0.015573	0.008796	0.015832	0.007133
Item_Type	1.000000	0.215021	1.000000	0.008476	0.007095	0.006881	0.011276	0.029976
Outlet_Identifier	0.102063	0.024022	0.008476	1.000000	1.000000	1.000000	1.000000	0.000275
Outlet_Size	0.073799	0.015573	0.007095	1.000000	1.000000	0.578765	0.552700	0.000094
Outlet_Location_Type	0.082409	0.008796	0.006881	1.000000	0.578765	1.000000	0.526540	0.000069
Outlet_Type	0.120311	0.015832	0.011276	1.000000	0.552700	0.526540	1.000000	0.000254
source	0.328087	0.007133	0.029976	0.000275	0.000094	0.000069	0.000254	1.000000

***Correlation Between Numeric with Categorical**

Outlet_Type	Grocery Store	Supermarket Type1	Supermarket Type2	Supermarket Type3
Item_Type				
Baking Goods	0.116306	0.066692	0.068206	0.066524
Breads	0.111612	0.067579	0.067976	0.066038
Breakfast	0.126056	0.079993	0.083079	0.078609
Canned	0.112484	0.066167	0.064655	0.067549
Dairy	0.120742	0.069299	0.069000	0.069213
Frozen Foods	0.115895	0.065449	0.064184	0.064816
Fruits and Vegetables	0.114820	0.067123	0.067503	0.066758
Hard Drinks	0.109913	0.064873	0.063611	0.065297
Health and Hygiene	0.098924	0.055021	0.053081	0.054924
Household	0.099288	0.057774	0.057858	0.057625
Meat	0.095960	0.058676	0.058574	0.058451
Others	0.096714	0.055688	0.056357	0.055413
Seafood	0.122593	0.067158	0.071047	0.075858
Snack Foods	0.114660	0.065578	0.065584	0.065126
Soft Drinks	0.108376	0.064303	0.064451	0.065485
Starchy Foods	0.117891	0.066730	0.068340	0.067608

Part 4: Prepare the data

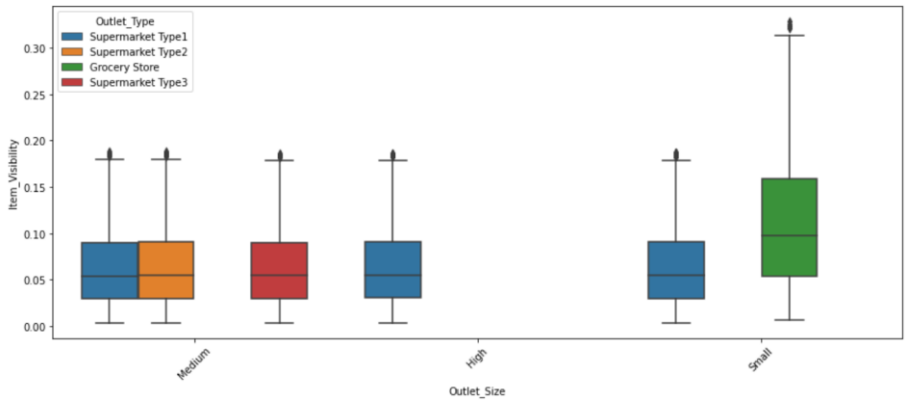
4.1. Skewness/Inconsistent/Missing/Outlier Handling

- *Item_Visibility :* As seen earlier visibilty has 0 values which is not possible as a product will have some visibility in marts

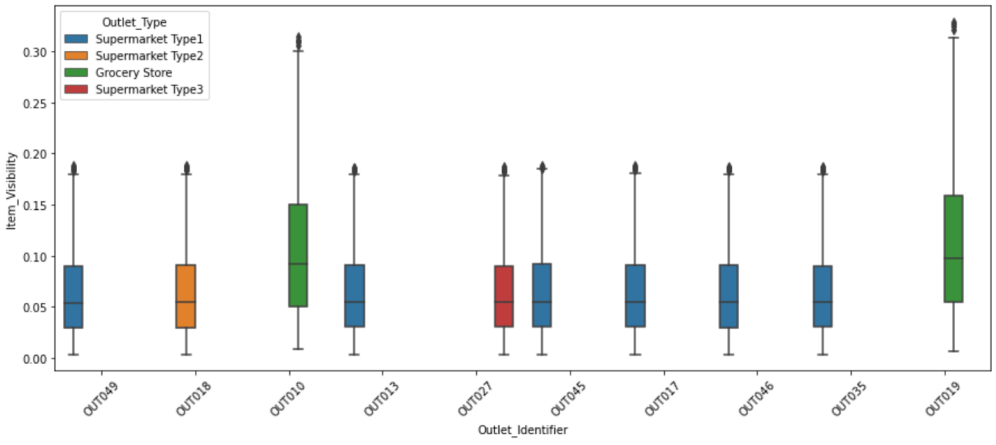
```
0.000000    879
0.076841      3
0.077011      3
0.077169      3
0.076792      3
           ...
0.162572      1
0.014826      1
0.058034      1
0.043786      1
0.104720      1
Name: Item_Visibility, Length: 13006, dtype: int64
```

- *Checking visibility in Outlet size with Outlet type*: Here visibility contains missing values

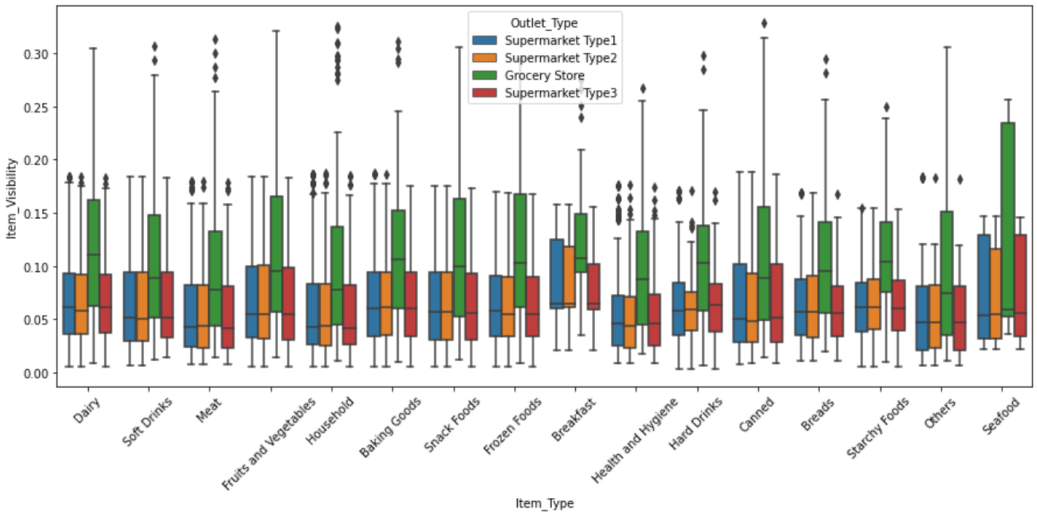
```
(array([0, 1, 2]),
 [Text(0, 0, 'Medium'), Text(1, 0, 'High'), Text(2, 0, 'Small')])
```



- *Checking visibility in Outlet identifier with Outlet type:* This also contain missing values



- *Checking visibility in Item type with Outlet type:* Here thee are no missing values for missing visibility hence this can be used to fill nan values in visibility column



- *Creating pivot table to help fill nan values of visibility from here:*

4.1.1. Item_Weight

- We try to fill the nan values of weight by using values from item identifier

Item_Identifier	Item_Weight
DRA12	11.600
DRA24	19.350
DRA59	8.270
DRB01	7.390
DRB13	6.115
...	...
NCZ30	6.590
NCZ41	19.850
NCZ42	10.500
NCZ53	9.600
NCZ54	14.650

1559 rows × 1 columns

4.1.2. Outlet_Size

- Grocery stores are usually smaller than super markets so we will replace NaN values by small

```
NaN       4016
Medium    4655
Small     3980
High      1553
Name: Outlet_Size, dtype: int64
```

Outlet_Type	Outlet_Size
Grocery Store	Unknown
Supermarket Type1	Small
Supermarket Type2	Medium
Supermarket Type3	Medium

- Filling nan values with mode : No null values remain

```
df.isnull().sum()
✓ 0.1s

Item_Identifier              0
Item_Weight                  0
Item_Fat_Content             0
Item_Visibility              0
Item_Type                    0
Item_MRP                     0
Outlet_Identifier            0
Outlet_Establishment_Year    0
Outlet_Size                  0
Outlet_Location_Type         0
Outlet_Type                  0
Item_Outlet_Sales            0
source                       0
dtype: int64
```

4.1.3. Item_Identifier

- Now the item identifier column looks more meaningful so we will keep it.

```
array(['FDA', 'DRC', 'FDN', 'FDX', 'NCD', 'FDP', 'FDO', 'FDH',
'FDU',
       'FDY', 'FDS', 'FDF', 'NCB', 'DRI', 'FDW', 'FDC', 'FDR',
'FDV',
       'DRJ', 'FDE', 'NCS', 'DRH', 'NCX', 'DRZ', 'FDB', 'FDK',
'FDL',
       'FDM', 'NCP', 'NCL', 'DRK', 'FDI', 'FDZ', 'NCI', 'FDJ',
'FDG',
       'NCZ', 'FDQ', 'FDD', 'DRG', 'NCR', 'FDT', 'DRB', 'DRE',
'DRA',
       'NCF', 'NCH', 'NCO', 'NCN', 'NCC', 'DRD', 'DRF', 'DRL',
'NCM',
       'NCU', 'DRY', 'NCW', 'DRM', 'NCT', 'NCQ', 'DRP', 'DRQ',
'NCK',
       'NCY', 'DRN', 'NCA', 'NCE', 'NCJ', 'NCV', 'NCG', 'DRO'],
      dtype=object)
```

- Correcting year column by subtracting it from 2021

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location
0	FDA	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	22.0	Medium	
1	DRC	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	12.0	Medium	
2	FDN	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	22.0	Medium	
3	FDX	19.20	Regular	0.114820	Fruits and Vegetables	182.0950	OUT010	23.0	Small	
4	NCD	8.93	Low Fat	0.057774	Household	53.8614	OUT013	34.0	High	

Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales	source
Tier 1	Supermarket Type1	3735.1380	train
Tier 3	Supermarket Type2	443.4228	train
Tier 1	Supermarket Type1	2097.2700	train
Tier 3	Grocery Store	732.3800	train
Tier 3	Supermarket Type1	994.7052	train

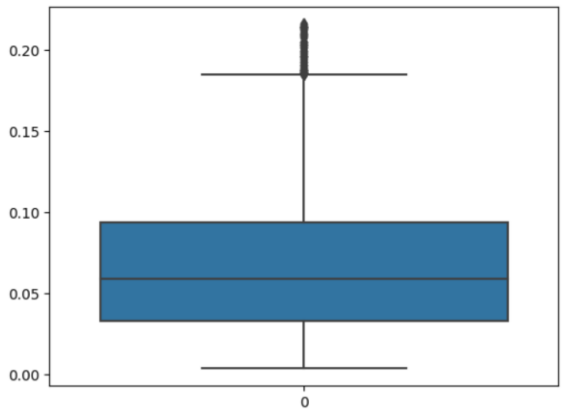
- Merging all the low fat categories to Low fat and regular categories to Regular

4.2. Encoding categorical features

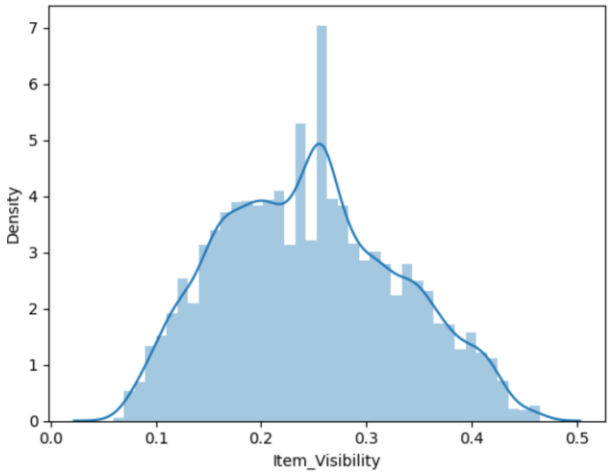
```
Item_Identifier              float64
Item_Weight                  float64
Item_Fat_Content             float64
Item_Visibility              float64
Item_Type                    float64
Item_MRP                     float64
Outlet_Identifier            float64
Outlet_Establishment_Year    float64
Outlet_Size                  float64
Outlet_Location_Type         float64
Outlet_Type                  float64
Item_Outlet_Sales            float64
source                        object
dtype: object
```

All columns are converted into float type except for source.

4.3. Removing outliers from Visibilitiy column



4.4. Removing skewness from visibility column



4.5. Feature scaling (Normalization & Standardization)

4.6. Separating the data into train and test

```
train
✓ 0.1s
```

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type	Item_Outlet_Sales
0	0.271429	0.282525	0.0	0.164962	0.266667	0.927507	1.000000	0.416667	0.5	0.0	0.333333	3735.1380
1	0.028571	0.081274	1.0	0.194970	0.933333	0.072068	0.333333	0.000000	0.5	1.0	0.666667	443.4228
2	0.457143	0.770765	0.0	0.171824	0.666667	0.468288	1.000000	0.416667	0.5	0.0	0.333333	2097.2700
3	0.600000	0.871986	1.0	0.688223	0.400000	0.640093	0.000000	0.458333	1.0	1.0	0.000000	732.3800
4	0.685714	0.260494	0.0	0.445327	0.600000	0.095805	0.111111	0.916667	0.0	1.0	0.333333	994.7052
...	...	...	...	...	...	...	...	...	...	...	...	...
8420	0.342857	0.137541	0.0	0.440225	0.866667	0.777729	0.111111	0.916667	0.0	1.0	0.333333	2778.3834
8421	0.528571	0.227746	1.0	0.387107	0.000000	0.326263	0.777778	0.291667	1.0	0.5	0.333333	549.2850
8422	0.771429	0.359929	0.0	0.315158	0.533333	0.228492	0.666667	0.208333	1.0	0.5	0.333333	1193.1136
8423	0.457143	0.158083	1.0	0.792362	0.866667	0.304939	0.333333	0.000000	0.5	1.0	0.666667	1845.5976
8424	0.085714	0.610003	0.0	0.375000	0.933333	0.187510	0.888889	0.500000	1.0	0.0	0.333333	765.6700

8425 rows × 12 columns

```
test
✓ 0.1s
```

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_Location_Type	Outlet_Type
0	0.585714	0.964275	0.0	0.067049	0.866667	0.325012	1.000000	0.416667	0.5	0.0	0.333333
1	0.585714	0.222983	1.0	0.335997	0.266667	0.237819	0.222222	0.083333	1.0	0.5	0.333333
2	0.828571	0.598095	0.0	0.630769	0.733333	0.893316	0.000000	0.458333	1.0	1.0	0.000000
3	0.500000	0.164335	0.0	0.158480	0.866667	0.525233	0.222222	0.083333	1.0	0.5	0.333333
4	0.614286	0.538553	1.0	0.701863	0.266667	0.861381	0.555556	1.000000	0.5	1.0	1.000000
...	...	...	...	...	...	...	...	...	...	...	...
5593	0.285714	0.353974	1.0	0.139058	0.866667	0.467004	0.888889	0.500000	1.0	0.0	0.333333
5594	0.314286	0.181304	1.0	0.785119	1.000000	0.585126	0.333333	0.000000	0.5	1.0	0.666667
5595	0.842857	0.324204	0.0	0.521286	0.533333	0.371199	0.777778	0.291667	1.0	0.5	0.333333
5596	0.400000	0.639774	1.0	0.486929	0.200000	0.778154	0.222222	0.083333	1.0	0.5	0.333333
5597	0.557143	0.294433	1.0	0.650621	0.200000	0.205884	0.777778	0.291667	1.0	0.5	0.333333

5598 rows × 11 columns

Part 5. Train and evaluate models

5.1. Modelling Phase

1. Importing necessary modules
2. Importing models
3. Choosing the best random state using Logistic regression
4. Creating list of models and another list mapped to their names
5. Creating models
6. Calculating scores of the model and appending them to a list
7. Creating Dataframe

Model	Mean Absolute Error	Mean Squared Error	Root Mean Squared Error	R2 Score	Mean of Cross validaton Score
Kneighbors Regressor	809.6622	1.301844e+06	1140.9836	26.14	50.28
Linear Regression	886.4671	1.370332e+06	1170.6120	13.49	49.79
Lasso	885.4656	1.368904e+06	1170.0017	12.70	49.82
Ridge	886.2023	1.369934e+06	1170.4417	13.16	49.79
Elastic Net	1212.1723	2.344823e+06	1531.2815	-7167.95	12.60

Decision Tree Regressor	1099.7594	2.518209e+06	1586.8865	20.70	16.26
Random Forest Regressor	779.8128	1.238802e+06	1113.0150	35.70	54.70
AdaBoost Regressor	1054.1472	1.693790e+06	1301.4571	-27.04	45.14
Gradient Boosting Regressor	742.6823	1.119748e+06	1058.1814	37.77	58.73
XGB Regressor	808.1595	1.307496e+06	1143.4578	33.33	51.57

From above analysis only Random Forest, Gradient Boost and Xgboost perform well with r2 score more than 51 and mean absolute error less than 810. Though the results are not as good therefore we further try to increase the scores by Feature Selection.

5.2. Feature Selection

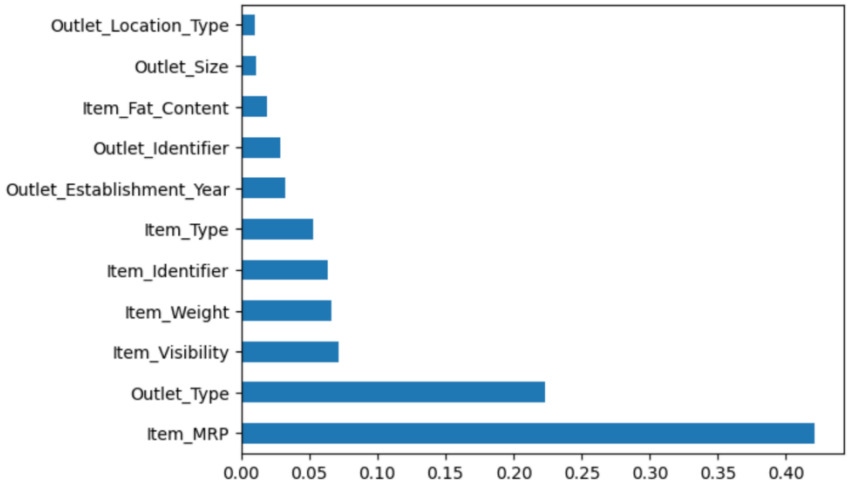
5.2.1. Using ANOVA test

- Below are the best features with there score in decreasing order after going through anova test

	Features	Score
5	Item_MRP	3.497038
10	Outlet_Type	2.105619
7	Outlet_Establishment_Year	1.213252
4	Item_Type	1.195552
6	Outlet_Identifier	1.190374
3	Item_Visibility	1.167628
2	Item_Fat_Content	1.164510
1	Item_Weight	1.135301
0	Item_Identifier	1.129296
8	Outlet_Size	1.059711
9	Outlet_Location_Type	1.037075

5.2.2. Using feature importance of Extra trees regressor

- Plot graph of feature importance for better visualization



⇨ MRP, Outlet weight are the most important features and the list follows. Above list contains features in order with most important feature on the top and least important feature below.

5.2.3. Using Lasso Coefficient

	Features	Coefficients
5	Item_MRP	3689.411377
10	Outlet_Type	2483.451777
6	Outlet_Identifier	482.277225
3	Item_Visibility	260.407711
8	Outlet_Size	248.076053
9	Outlet_Location_Type	231.696856
2	Item_Fat_Content	54.623804
7	Outlet_Establishment_Year	7.023098
0	Item_Identifier	1.390688
1	Item_Weight	0.398274
4	Item_Type	0.000000

⇨ Above dataframe shows features with their coefficients values. Item Type has coefficient 0 signifying that it is least important feature according to Lasso

5.3. Conclusion

- Anova test and feature importance tell us that Outlet_Location_Type is the least significant feature. Feature selection tells us that Item_Type is the least significant feature. So we create 3 set of data set:

- x1 - After removing Outlet_Location_Type
- x2 - After removing Item_Type
- x3 - After removing both Outlet_Location_Type and Item_Type.

From above analysis we see that previous models which were performance well are the one performing well even this time and the dataset providing least Root errors and highest mean cross validation score is Dataset x2 which we get after removing Item_Type which we got using feature selection of Lasso, even though results do not vary much even after doing feature selection so we will keep the original dataset for training purposes without loosing any data.

```
#Testing x1 dataset
createmodels(models,x1,y,names)
```
✓ 51.9s

Model	Mean Absolute Error	Mean Squared Error	Root Mean Squared Error	R2 Score	Mean of Cross validaton Score
KNeighborsRegressor	808.3956	1.301318e+06	1140.7533	26.37	50.36
LinearRegression	887.1898	1.373900e+06	1172.1349	13.30	49.72
Lasso	886.1188	1.372222e+06	1171.4188	12.52	49.75
Ridge	886.8998	1.373430e+06	1171.9341	12.97	49.72
ElasticNet	1213.0091	2.350025e+06	1532.9791	-7719.87	12.42
DecisionTreeRegressor	1080.0461	2.411496e+06	1552.8991	20.63	17.08
RandomForestRegressor	780.5690	1.233084e+06	1110.4432	35.52	54.64
AdaBoostRegressor	1110.1949	1.809982e+06	1345.3555	-62.41	43.59
GradientBoostingRegressor	743.0533	1.121204e+06	1058.8690	37.69	58.72
XGBRegressor	806.4428	1.312965e+06	1145.8470	33.49	51.65


```
#Testing x1 dataset
createmodels(models,x1,y,names)
```
✓ 51.9s

Model	Mean Absolute Error	Mean Squared Error	Root Mean Squared Error	R2 Score	Mean of Cross validaton Score
KNeighborsRegressor	808.3956	1.301318e+06	1140.7533	26.37	50.36
LinearRegression	887.1898	1.373900e+06	1172.1349	13.30	49.72
Lasso	886.1188	1.372222e+06	1171.4188	12.52	49.75
Ridge	886.8998	1.373430e+06	1171.9341	12.97	49.72
ElasticNet	1213.0091	2.350025e+06	1532.9791	-7719.87	12.42
DecisionTreeRegressor	1080.0461	2.411496e+06	1552.8991	20.63	17.08
RandomForestRegressor	780.5690	1.233084e+06	1110.4432	35.52	54.64
AdaBoostRegressor	1110.1949	1.809982e+06	1345.3555	-62.41	43.59
GradientBoostingRegressor	743.0533	1.121204e+06	1058.8690	37.69	58.72
XGBRegressor	806.4428	1.312965e+06	1145.8470	33.49	51.65


```
#Testing x3 dataset
createmodels(models,x3,y,names)
```
✓ 49.5s

Model	Mean Absolute Error	Mean Squared Error	Root Mean Squared Error	R2 Score	Mean of Cross validaton Score
KNeighborsRegressor	805.1753	1.275904e+06	1129.5593	30.07	50.76
LinearRegression	887.1800	1.373886e+06	1172.1289	13.30	49.74
Lasso	886.1188	1.372222e+06	1171.4188	12.52	49.76
Ridge	886.8925	1.373416e+06	1171.9284	12.97	49.74
ElasticNet	1213.0930	2.350185e+06	1533.0314	-7726.04	12.42
DecisionTreeRegressor	1088.5956	2.444907e+06	1563.6197	21.99	14.75
RandomForestRegressor	785.5638	1.256878e+06	1121.1059	34.74	54.71
AdaBoostRegressor	989.3338	1.536632e+06	1239.6097	-11.64	44.93
GradientBoostingRegressor	742.7088	1.123102e+06	1059.7651	37.39	58.69
XGBRegressor	806.1502	1.328329e+06	1152.5317	32.78	52.32

⇨ After Hyperparameter tunning the best model with least error and highest r2 score and cross validation score is Gradient Boost

Part 6. Fine-tune models

6.1. Random Forest

```
RandomForestRegressor(max_depth=6, min_samples_leaf=4, min_samples_split=3)
{'max_depth': 6, 'min_samples_leaf': 4, 'min_samples_split': 3, 'n_estimators': 100}
0.5939707991358253
```

```
Mean Absolute Error is 740.9997
Mean Squared Error is 1122815.3105
Root Mean Squared Error is 1059.6298
R2 Score is 37.88
Mean of cross validaton Score is 59.0571
```

6.2. Gradient Boost

```
RandomizedSearchCV(cv=5, estimator=GradientBoostingRegressor(),
                   param_distributions={'learning_rate': [0.05, 0.1],
                                        'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9,
                                                      10],
                                        'n_estimators': [100, 200, 300, 400,
                                                         500],
                                        'subsample': [0.5, 1]})
```

```
Mean Absolute Error is 743.094
Mean Squared Error is 1108163.5442
Root Mean Squared Error is 1052.6935
R2 Score is 36.08
Mean of cross validaton Score is 59.196
```

6.3. Xtreme Gradient Boost

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.7, gamma=0.1, gpu_id=-1,
             importance_type='gain', interaction_constraints='',
             learning_rate=0.05, max_delta_step=0, max_depth=5,
             min_child_weight=3, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=4, num_parallel_tree=1, random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
             tree_method='exact', validate_parameters=1, verbosity=None)
{'min_child_weight': 3, 'max_depth': 5, 'learning_rate': 0.05, 'gamma': 0.1, 'colsample_bytree': 0.7}
0.5878173718244176
```

```
Mean Absolute Error is 746.9179
Mean Squared Error is 1132742.7915
Root Mean Squared Error is 1064.3039
R2 Score is 32.45
Mean of cross validaton Score is 58.6744
```

⇨ Conclusion After Hyperparameter tunning the best model with least error and highest r2 score and cross validation score is Gradient Boost

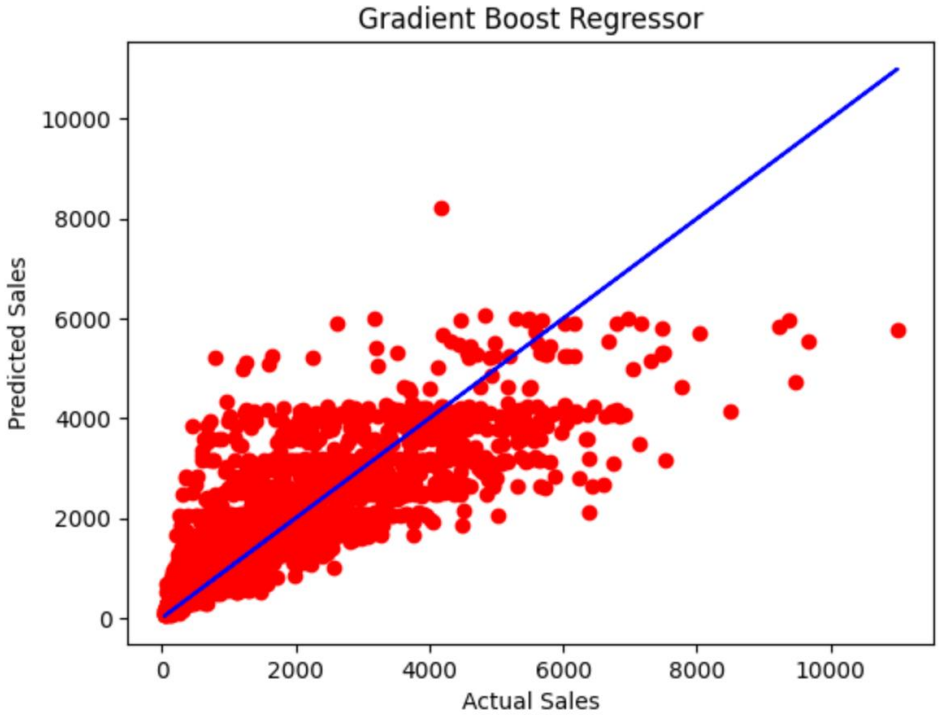
Part 7. Analyze and test your solution

7.1. Finalizing the best model

7.2. Evaluation Metrics

```
Mean Absolute Error is 739.775
Mean Squared Error is 1112466.5404
Root Mean Squared Error is 1054.7353
R2 Score is 34.69
Mean of cross validaton Score is 59.196
```

- Plot image



⇨ There are still a lot of outliers in our output

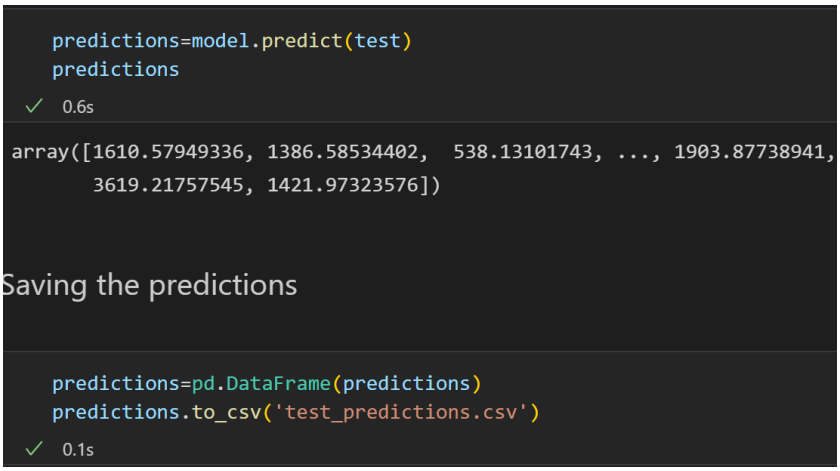
- Saving Model

```
import joblib
joblib.dump(model,'Mart_Sales.obj')
```

✓ 0.7s

```
['Mart_Sales.obj']
```

- Predicting model



REFERENCES

[1] Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media.

[2] Machine Learning (2020) IBM Cloud Education. Retrieve 11/12/2022 from https://www.ibm.com/cloud/learn/machine-learning

[3] *Machine Learning* (n.d.). Wikipedia, the free encyclopedia. Retrieve 04/12/2022 from https://wikipedia.org/wiki/Machine_learning

[4] PhD.Tran Nhat Quang (2022). *Lectures on Machine Learning.*

[5] *Ensemble Classifier- geeksforgeeks.*(n.d.). Retrieve 05/12/2022 from https://www.geeksforgeeks.org/ensemble-classifier-data-mining/

[6] SHIVAN KUMAR. (2021). Dataset: Big Mart Sales Prediction Datasets. Retrieve 4/12/2022 from https://www.kaggle.com/datasets/shivan118/big-mart-sales-prediction-datasets