Contents

Machine Learning Zoomcamp 2024-DataTalksClub

https://www.youtube.com/playlist?list=PL3MmuxUbc_hIhxl5Ji8t4O6lPAOpHaCLR
Alexey Grigorev based on his book O-Reilly Machine Learning Bookcamp

Follow the course videos/notes on the Github repo-
https://github.com/DataTalksClub/machine-learning-zoomcamp

Detailed Course Notes by Peter Ernicke on
https://knowmledge.com/category/courses/ml-zoomcamp/introduction/

Course Overview

Free 4-Month Course on ML Engineering

Self-Paced Learning

All course materials are freely available for independent study. Follow these steps:

1. Watch the course videos and work through the code.
2. Join the Slack community (`#course-ml-zoomcamp`).
3. Ask questions in Slack or refer to the FAQ.
4. Complete the homework assignments (solutions provided but attempt first).
5. Work on at least one project for deeper learning.

Syllabus Overview

The course consists of structured modules covering the full ML pipeline, from fundamentals to advanced techniques.

Prerequisites

- Prior programming experience (at least 1+ year)
- Comfort with command line basics
- No prior ML knowledge required

Module 1: Introduction to Machine Learning

- ML vs Rule-Based Systems
- Supervised Learning
- CRISP-DM Framework
- Model Selection Process
- Environment Setup
- Homework

Module 2: Machine Learning for Regression

- Car Price Prediction Project
- Exploratory Data Analysis
- Linear Regression Basics
- Feature Engineering & Regularization
- Homework

Module 3: Machine Learning for Classification

- Churn Prediction Project
- Feature Selection & Encoding
- Logistic Regression
- Model Interpretation
- Homework

Module 4: Evaluation Metrics

- Accuracy, Precision, Recall
- ROC Curves & AUC
- Cross-Validation
- Homework

Module 5: Deploying ML Models

- Saving & Loading Models
- Flask API Deployment
- Docker & Virtual Environments
- Cloud Deployment (AWS)
- Homework

Module 6: Decision Trees & Ensemble Learning

- Decision Trees
- Random Forest & Gradient Boosting
- Model Selection & Hyperparameter Tuning
- Homework

Module 7: Neural Networks & Deep Learning

- TensorFlow & Keras
- Convolutional Neural Networks
- Transfer Learning
- Model Optimization & Regularization
- Homework

Module 8: Serverless Deep Learning

- Introduction to Serverless
- AWS Lambda & TensorFlow Lite
- API Gateway
- Homework

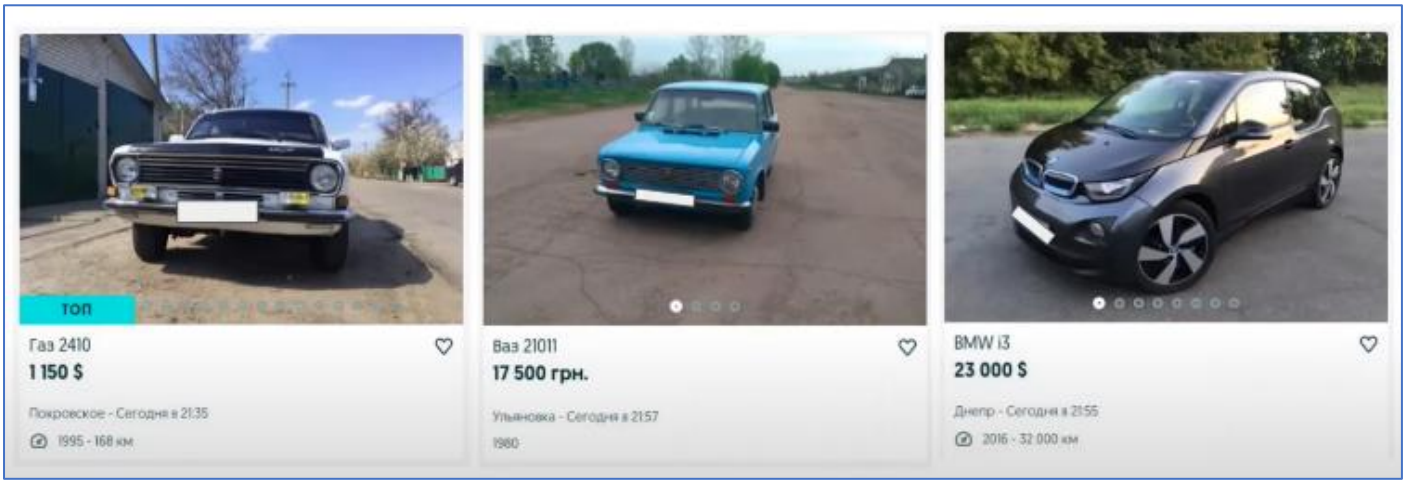
Module 9: Kubernetes & TensorFlow Serving

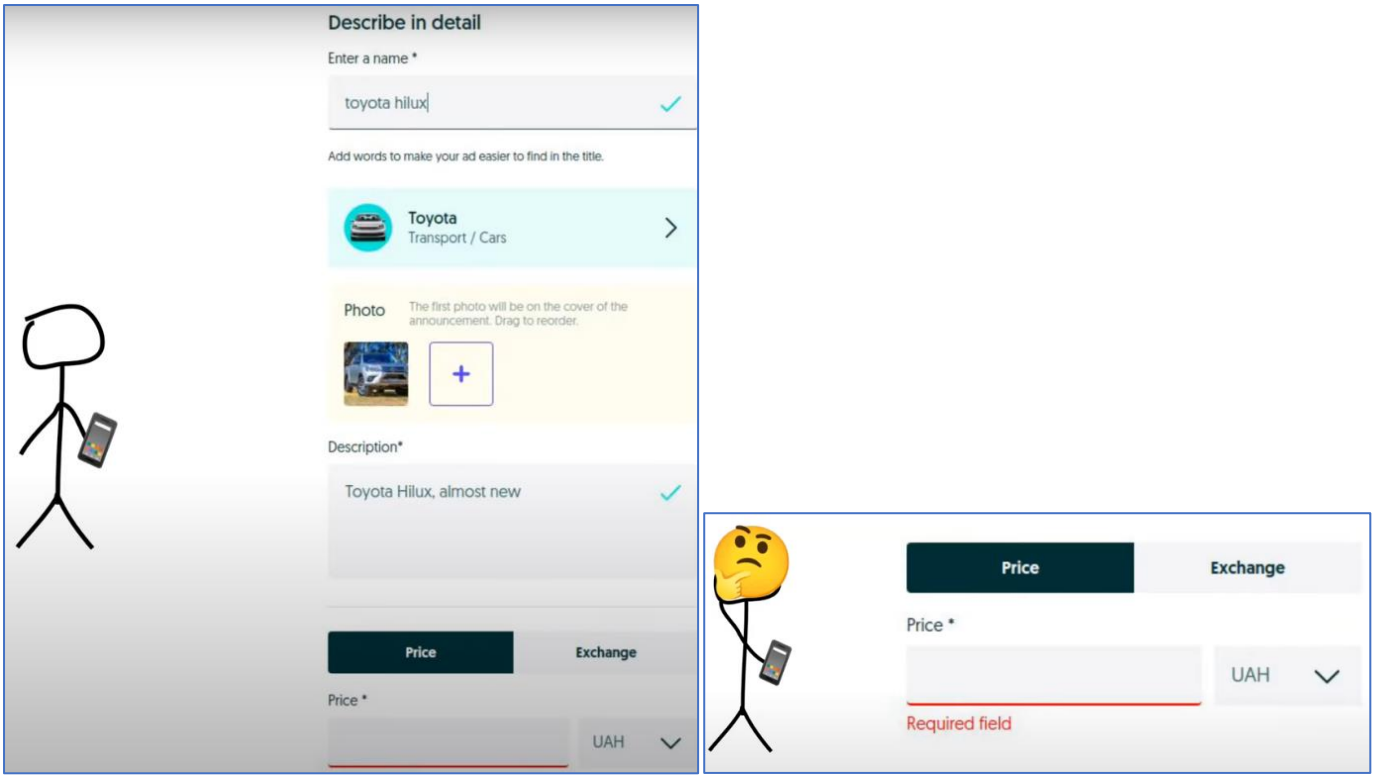
- TensorFlow Model Serving
- Kubernetes Basics
- Deploying ML Models to Kubernetes
- Homework

Midterm Project, 2 Capstone Projects

1. Introduction to Machine Learning

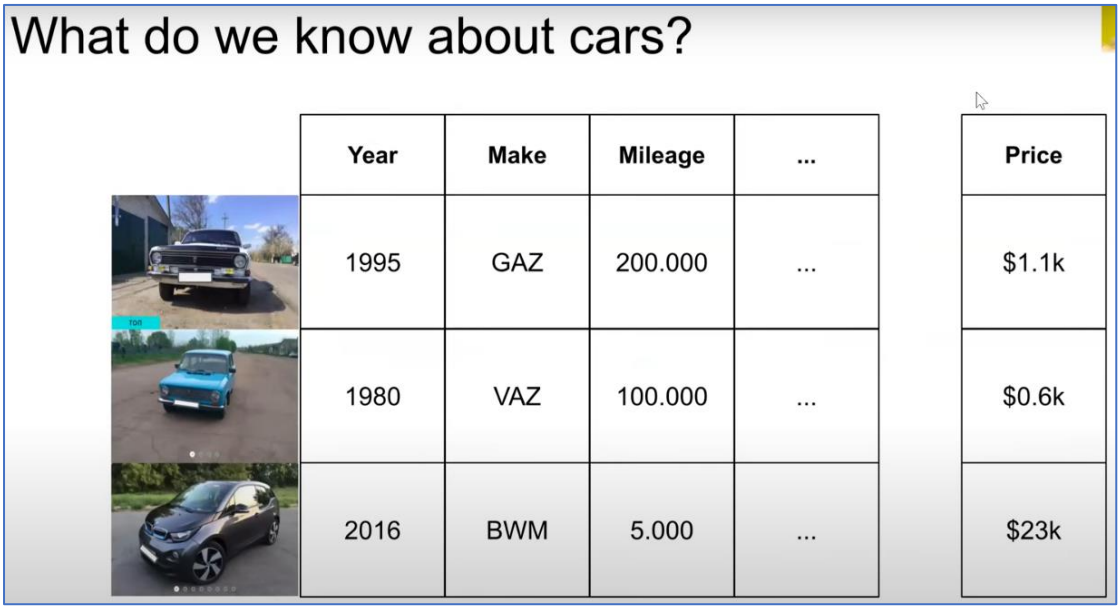
Imagine you have a Car Classifieds site where users can post ads about the cars that they want to sell. To post an ad, they provide details about the Car like Make/Model, Year, Expected Price.



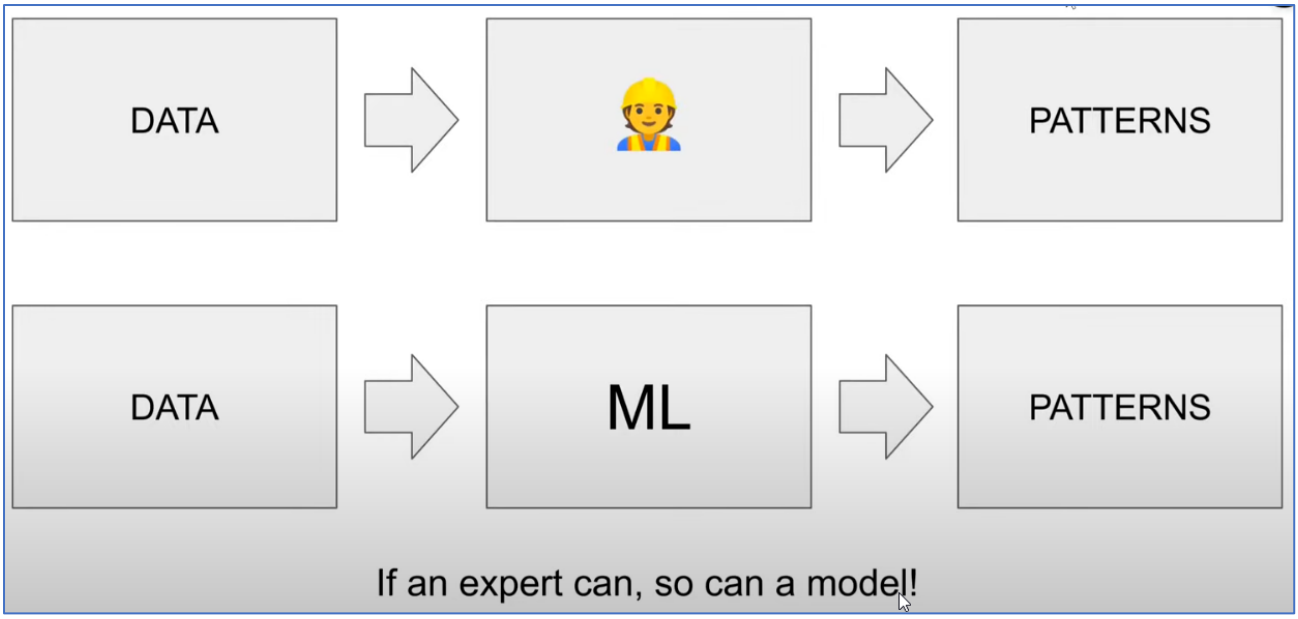


How does user arrive at the Expected price ? They want to put a price which is not too high that there will be no buyer and neither too low that they will make a loss. One way would be that they manually analyze other ads that have been sold to infer a rough estimate of the Expected Price - but this is too time-consuming and laborious work
Also, if our website could recommend expected price to the user, that would be a great differentiator to attract users to our site. This can be done through Machine Learning

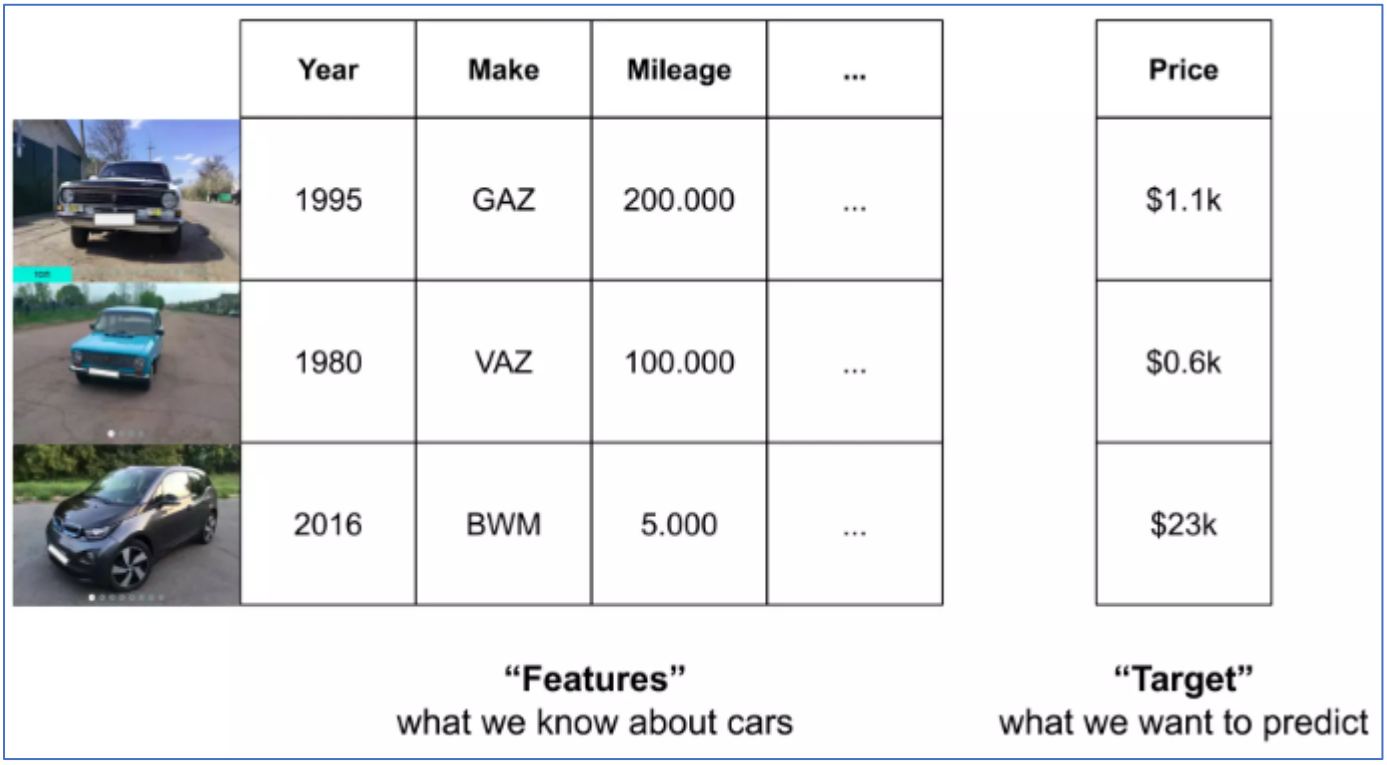
Thought process



From the data that we have on the site, we have many data points like Year of manufacture, Make, Mileage etc and their Prices
If you were an offline dealer, you would have some experts who will use these data points to quote an expected price. So, from the data, the experts will recognize/extract the patterns to predict a price range. And if a model can be taught to recognize/extract patterns, the model can predict the price range similar to the expert.



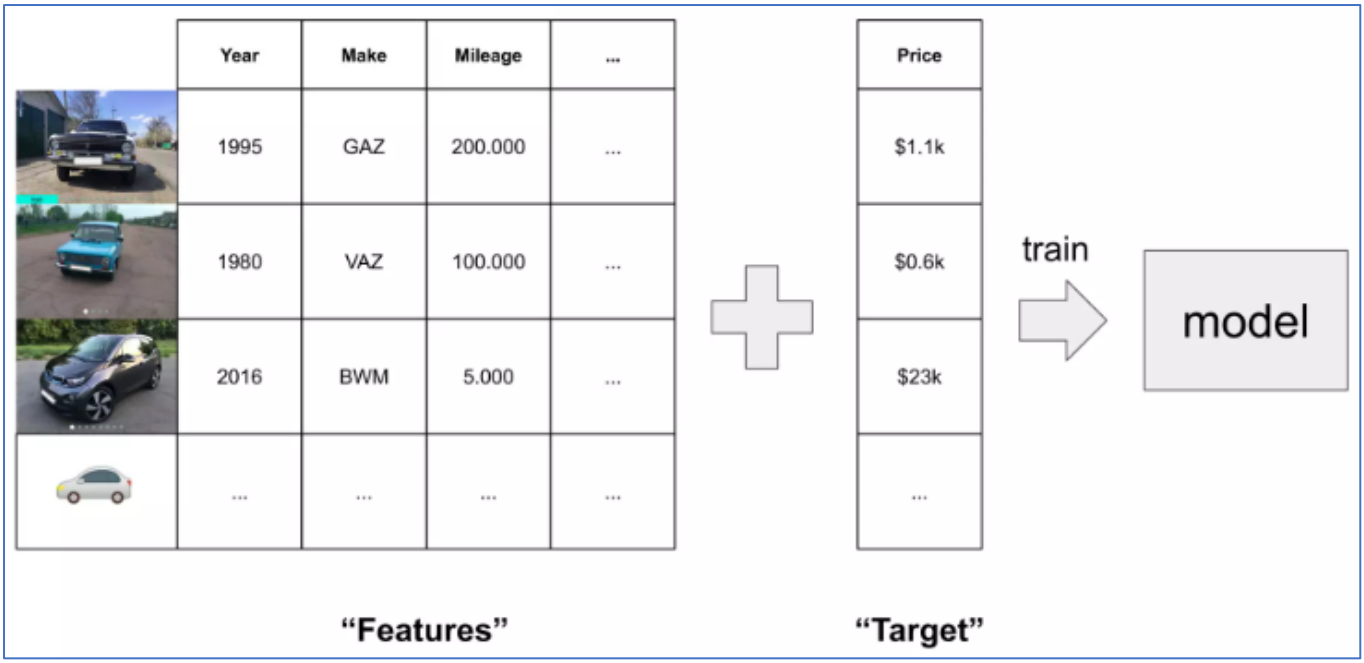
Basic entities in ML

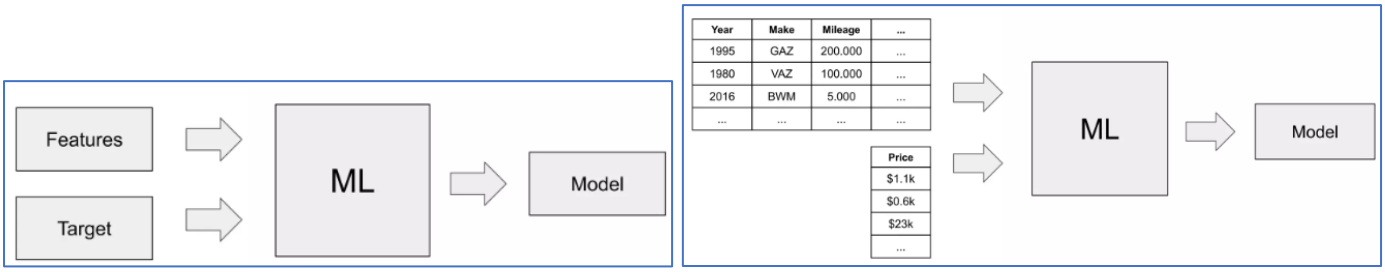


Features - the attributes that we know. In this case, Year of Mfg, Make, Mileage etc

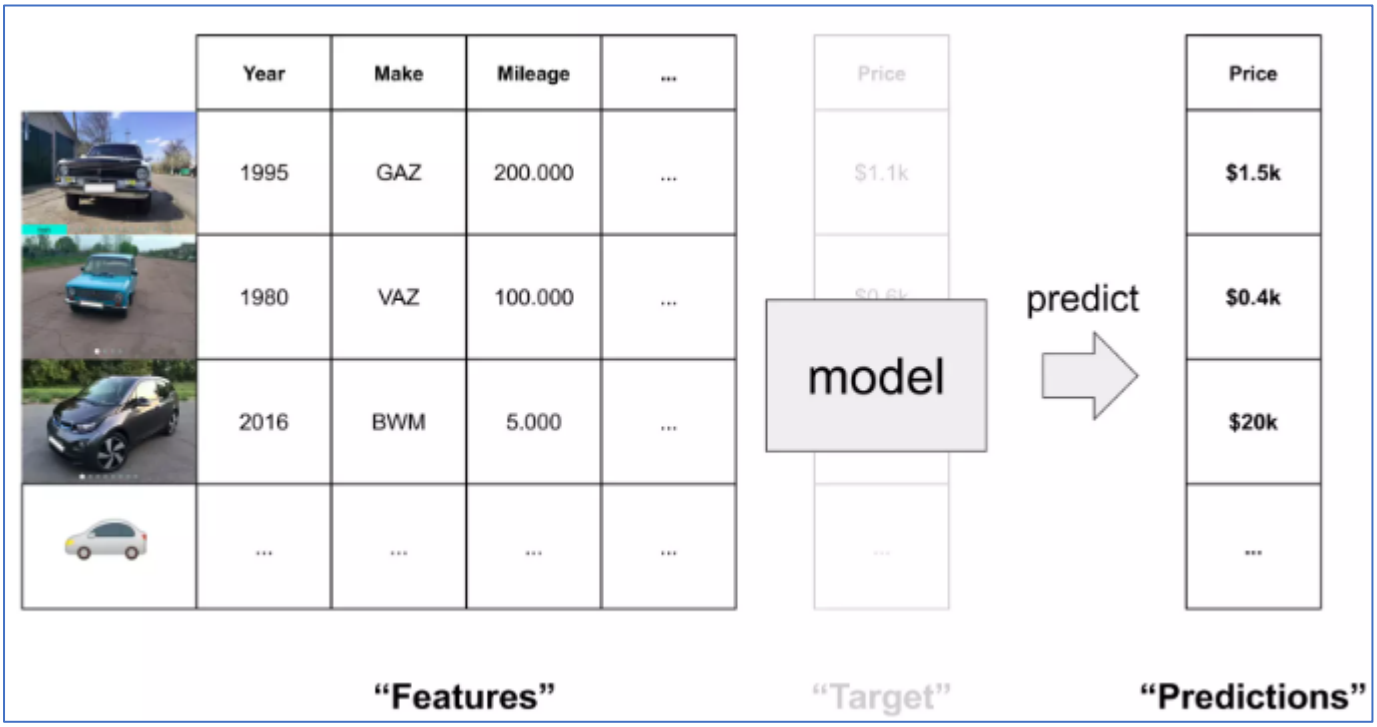
Target - the attribute that we want to predict based on the features. In this case, expected Price

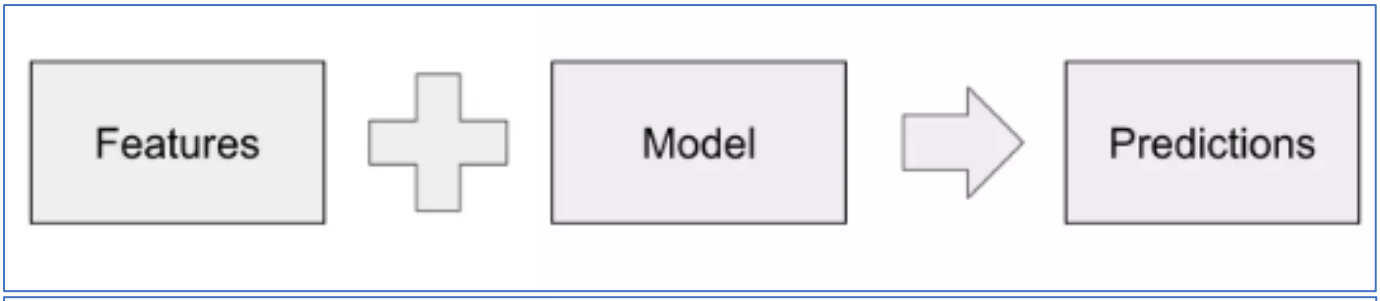
Training the Model - Machine Learning is nothing but training the Model using the Features and Target. The model encapsulates all the patterns it has learnt from the existing data

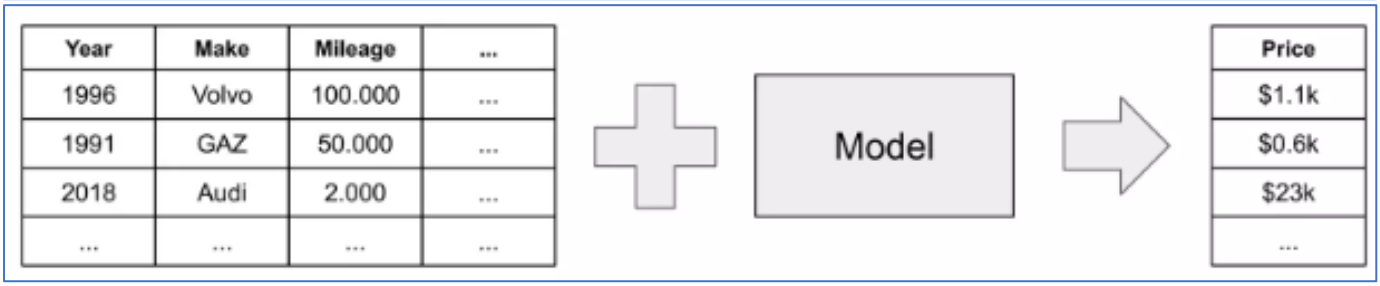




Using the model - Use the model to predict the Target for new Feature data. Often, the Predictions for existing Feature data will not be exact Target data but it is close





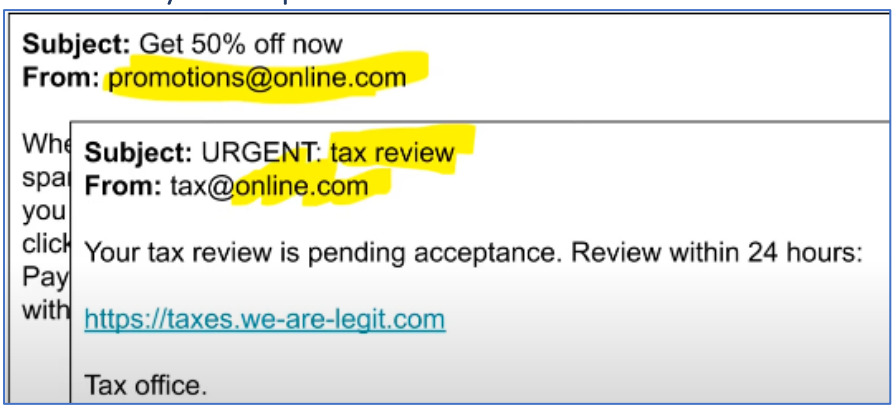


So, in our site, if user just enters the basic details about the car and our model can predict an expected price, the user would be very happy that they do not need to think about how to arrive at the expected price !!! User can ofcourse increase or decrease the predicted price but at least it gives them a starting point

ML vs Rule-based system

Lets understand this with simple example of detecting SPAM In email system.

Rule-based system implementation



You will try to define the rules from the existing emails like -

- If sender = promotions@online.com then “spam”
- If title contains “tax review” and sender domain is “online.com” then “spam”
- Otherwise, “good email”

and implement the rules in code-

```
def detect_spam(email):
    if email.sender == 'promotions@online.com':
        return SPAM
    if contains(email.title, ['tax', 'rewiew']) and
            domain(email.sender, 'online.com'):
        return SPAM
    return GOOD
```

This works for emails where these conditions are met but then you get newer emails which are SPAM and you check what new rule you can define.

Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

So, you identify that you can use the word "deposit" to define your new rule and update your code accordingly

- If sender = promotions@online.com then "spam"
- If title contains "tax review" and sender domain is "online.com" then "spam"
- **If body contains a word "deposit" then "spam"**
- Otherwise, "good email"

```
def detect_spam(email):
    if email.sender == 'promotions@online.com':
        return SPAM
    if contains(email.title, ['tax', 'rewiew']) and
            domain(email.sender, 'online.com'):
        return SPAM
    if contains(email.body, ['deposit']):
        return SPAM
    return GOOD
```

Again, this works for emails where these conditions are met but then some legitimate emails which are NOT SPAM are incorrectly tagged as SPAM. So, you check what additional rule you need to correct the errors-

Subject: Totally legit email
From: pedro@gmail.com

I transferred $50 to you one year ago, and now I'm moving out. Please refund my deposit.

Pedro.

- If sender = promotions@online.com then "spam"
- If title contains "tax review" and sender domain is "online.com" then "spam"
- If body contains a word "deposit"
  - **If sender domain is "test.com" then "spam"**
  - **If body >= 100 words then spam**
- Otherwise, "good email"

and you see this is a never-ending process of keeping to define and code the rules. So, here we can use MACHINE LEARNING instead

Machine Learning Implementation

Key steps involved here are -

1. Get the data - Have the user tag existing emails as SPAM/NOT SPAM
2. Define and calculate features - Start with some rules/conditions
3. Train and use the model - Use the features to train the model and use the model to classify SPAM/GOOD

Get the data

Getting data

SPAM

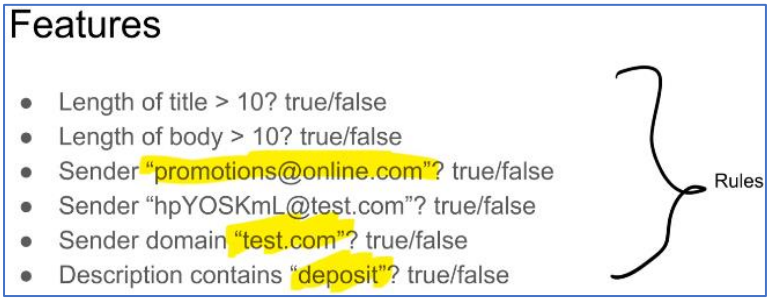
Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please deposit $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

So, the user can tag existing emails as SPAM/GOOD

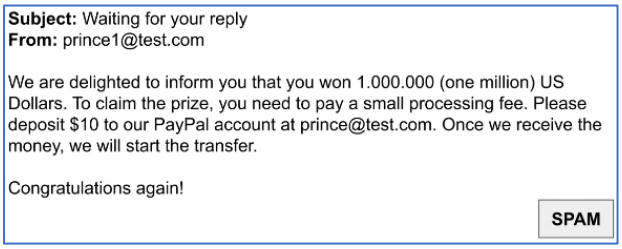
Define and calculate features

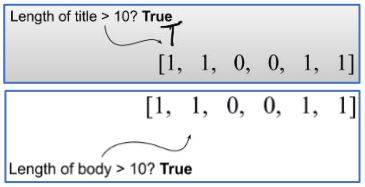


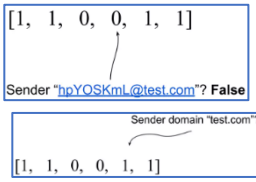
Start with rules and then use these rules as features

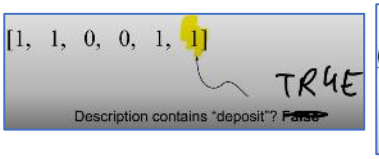
Use the rules as starting point to define features. Here, they are binary features (taking values of only 0 or 1)

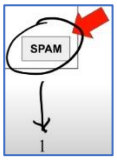
for each tagged email, calculate the values of the features and the outcome(Target) that user selected (SPAM or GOOD)











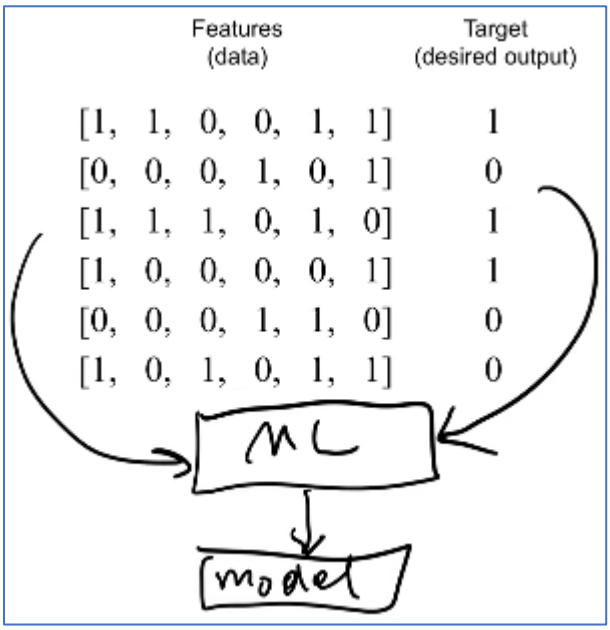
Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0

Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 1]	0

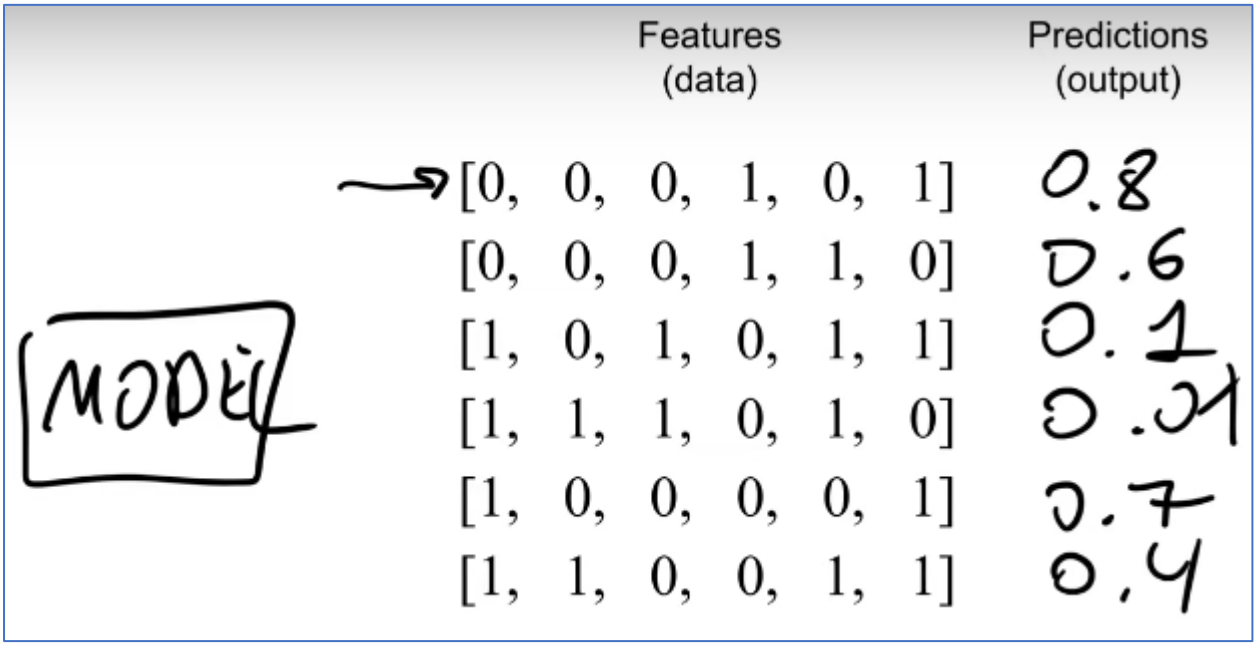
At the end of calculating features for all emails, you have the values for Features (Data) and Target (Desired Output)

Train the Model



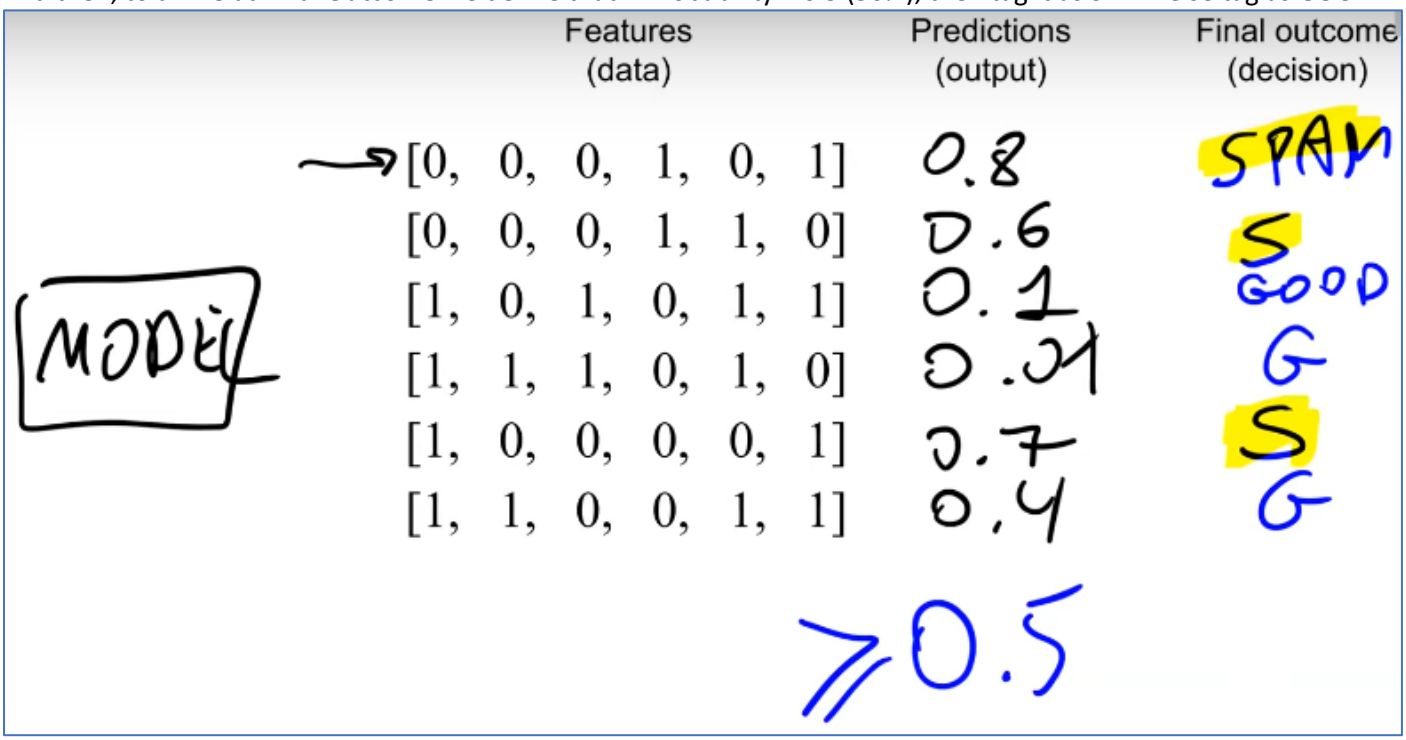
We feed the Features and Target to Machine Learning Algorithm to generate the model (fit the data)

Use the model

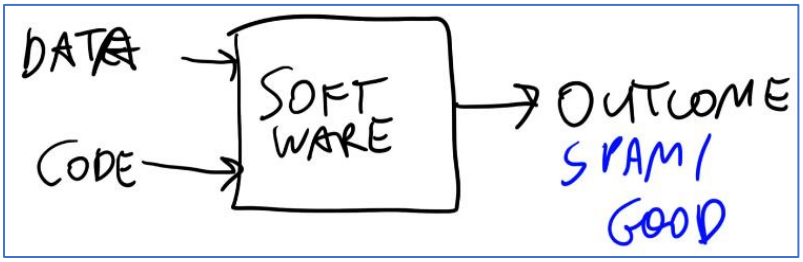


Then, we use the model on new data to generate Predictions which are generally probabilities (between 0 and 1). So, probability of first email is SPAM is 0.8 (80%), next is 0.6 (60%) and so on.

And then, to arrive at Final Outcome we define that if Probability > 0.5 (50%), then tag it as SPAM else tag as GOOD

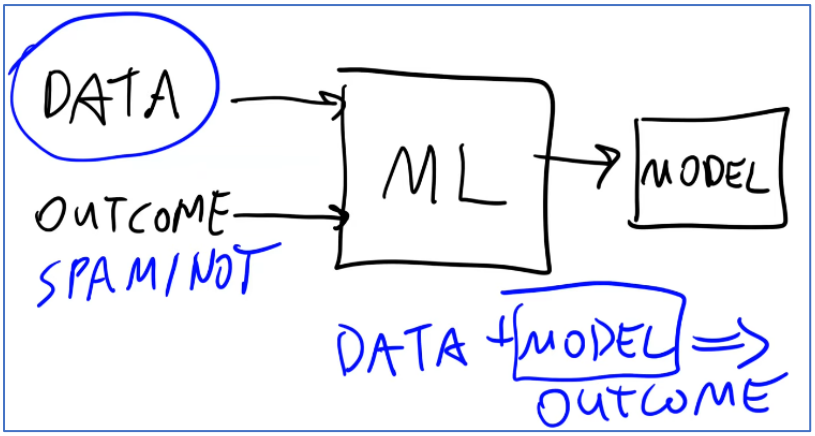


Difference between Rule-based and ML-based



data + code => software => outcome

In rule-based, we have hard-coded rules that acts on data to give the outcome



data + outcome => ML => model

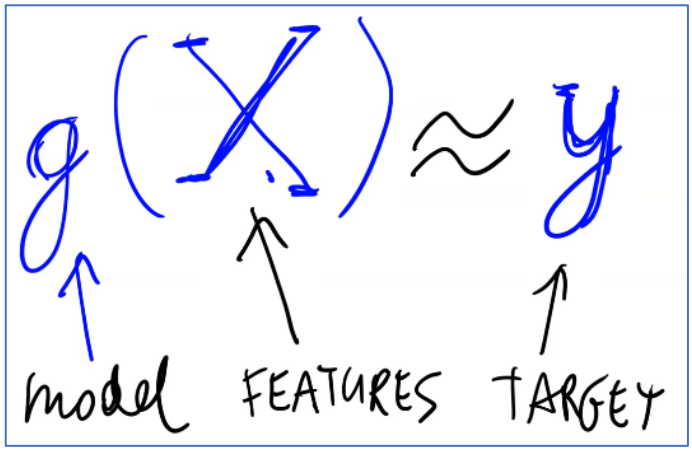
In ML-based, alongwith data we are inputting known outcome to the ML algorithm to extract pattern and generate a model. And then, we apply the model to new data to predict the outome

Supervised Machine Learning

Types of Supervised Learning - Regression, Classification,
Ranking (not covered in this course)

In the example of Car Price Prediction, we showed the ML model different cars and their prices and we did it for all the cars. In the example of SPAM classification also, we showed the ML model different examples of SPAM => We told the ML model i.e supervised the model to learn the pattern ➔ **SUPERVISED LEARNING**

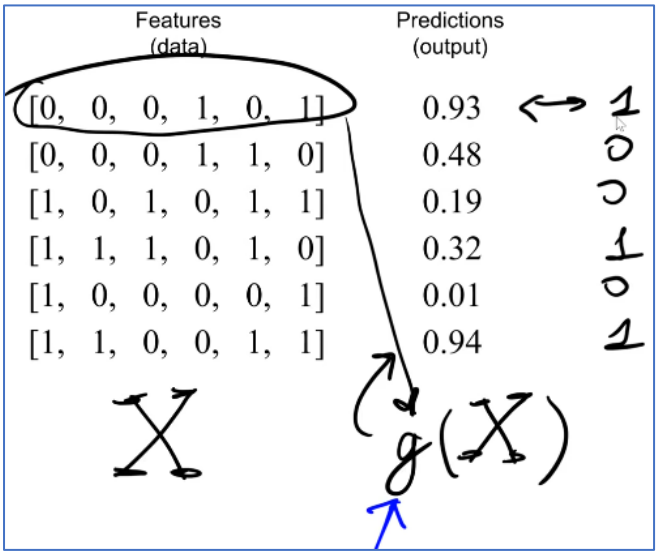
Features (data)	Target (desired output)
[1, 1, 0, 0, 1, 1]	1
[0, 0, 0, 1, 0, 1]	0
[1, 1, 1, 0, 1, 0]	1
[1, 0, 0, 0, 0, 1]	1
[0, 0, 0, 1, 1, 0]	0
[1, 0, 1, 0, 1, 1]	0



So, we have the feature values which is called **FEATURE MATRIX** denoted by capital X. It is a 2-dimensional array where rows represent the different examples and columns represent the feature values

We have the target values which is called the **TARGET VECTOR** (Desired Output) denoted by lowercase 'y'. It is a 1-dimensional array with one column which represents the desired output for each example

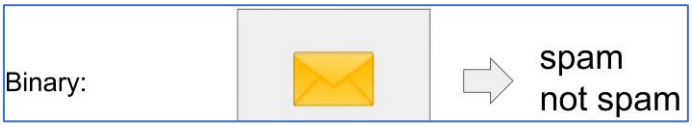
Now, by **TRAINING A MODEL** we mean a **FUNCTION g(X)** which when **applied on MATRIX X will give us output as close as possible to TARGET VECTOR y. g(X) is our MODEL**



In the case of SPAM classification, the predictions were probabilities and the target variable would have values only 0 and 1. So, the predictions are close to the target value

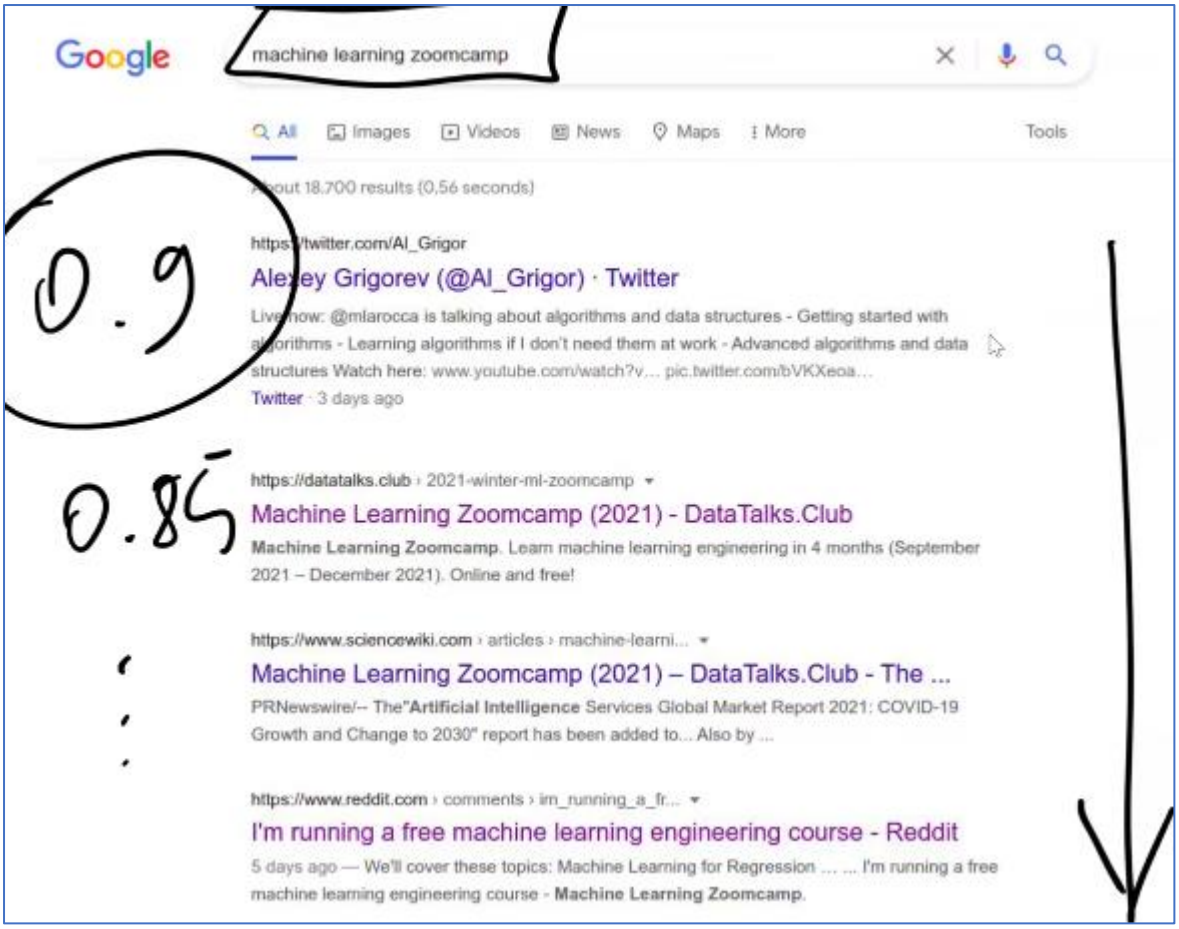
In **Regression** - g(X) returns a number to predict the value like price of car, house etc

In **Classification** - g(X) returns a category like whether the email is SPAM or not, image is of Dog or not.
Multiclass Classification: Classify the data into multiple different categories like whether image is a cat, dog or car
Binary Classification: Classify the data into only 2 categories like whether image is a cat or not, email is spam or not. Widely used



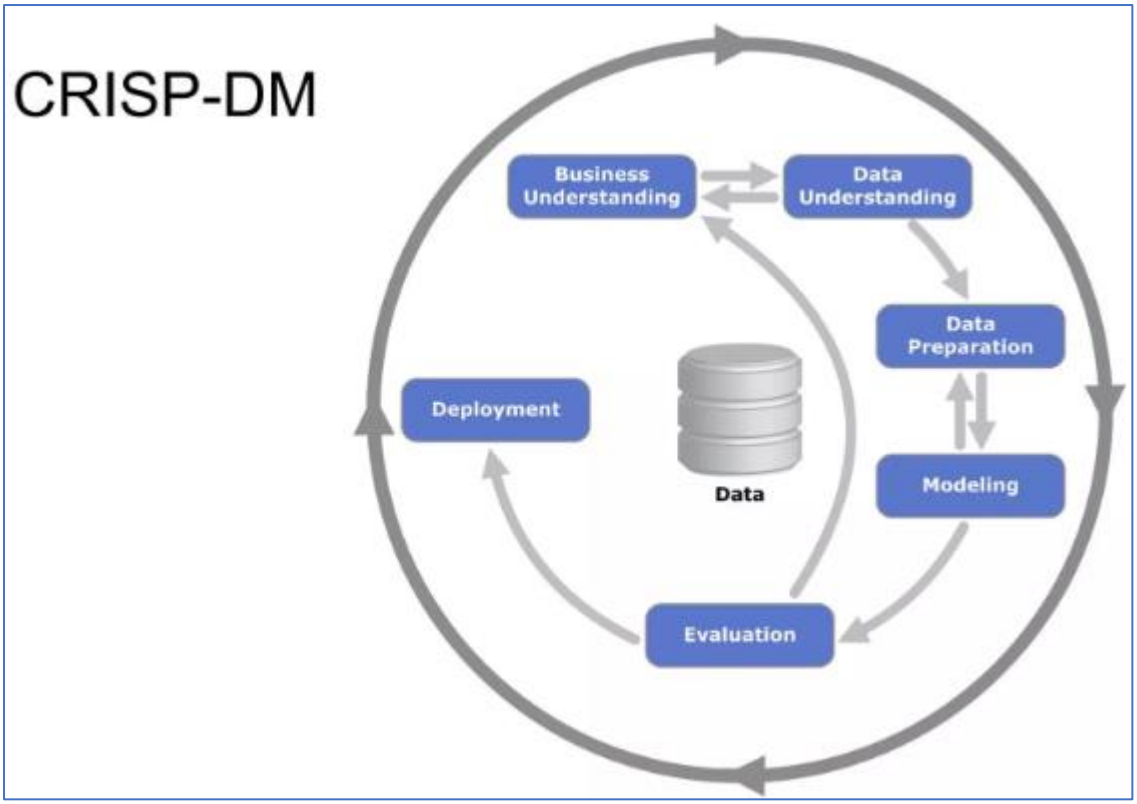
In **Ranking** - Used in Recommendation systems, like in Amazon on your homepage, you see top 6 most recommended products. g(X) function based on my profile, purchase history etc will try to score the products on the probability that I would buy them and rank them according to score between 0 -1. And then show me the top 6 products

Also, in Google search results, all the pages containing the search phrase are ranked between 0-1 based on relevance and probability that those pages are of my interest

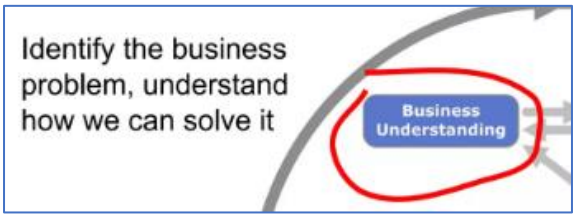


ML Project Lifecycle: CRISP-DM

CRISP-DM stands for Cross-Industry Standard Process for Data Mining, widely used framework for organizing and executing data mining projects, encompassing six phases: Business Understanding, Data Understanding, Data Preparation, Modeling, Evaluation, and Deployment. Formalized in 1999 by IBM but is still relevant today for ML projects



Business Understanding



Business understanding

- Our users complain about spam
- Analyze to what extent it's a problem
- Will Machine Learning help?
- If not: propose an alternative solution

Define the goal:

- Reduce the amount of spam messages, or
- Reduce the amount of complaints about spam

The goal has to be measurable

- Reduce the amount of spam by 50%

Data Understanding

Data understanding

Identify the data sources

- We have a report spam button
- Is the data behind this button good enough?
- Is it reliable?
- Do we track it correctly?
- Is the dataset large enough?
- Do we need to get more data?

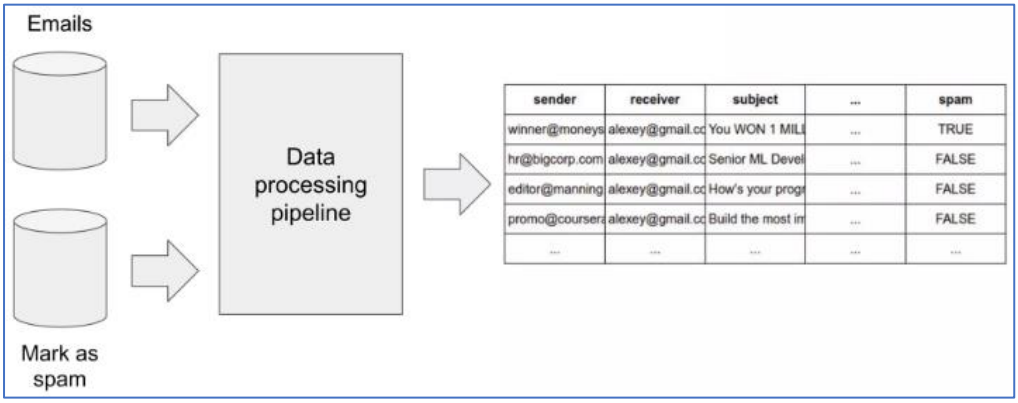
Identify the data sources

- It may influence the goal
- We may go back to the previous step and adjust it

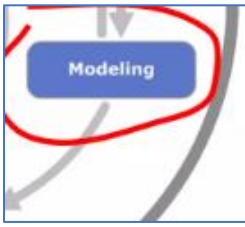
Data Preparation

Transform the data so it can be put into a ML algorithm

- Clean the data
- Build the pipelines
- Convert into tabular form



Modeling



Training the models: the actual Machine Learning happens here

Training a model:

- Try different models
- Select the best one

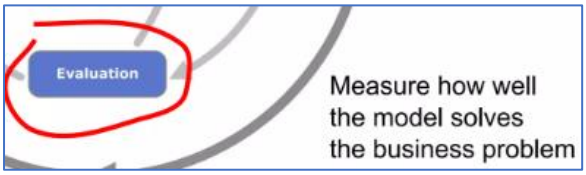
Which model to choose?

- Logistic regression
- Decision tree
- Neural network
- Or many others

Sometimes, we may go back to data preparation:

- Add new features
- Fix data issues

Evaluation



Is the model good enough?

- Have we reached the goal?
- Do our metrics improve?

Goal: Reduce the amount of spam by 50%

- Have we reduced it? By how much?
- (Evaluate on the test group)

Do a retrospective:

- Was the goal achievable?
- Did we solve/measure the right thing?

After that, we may decide to:

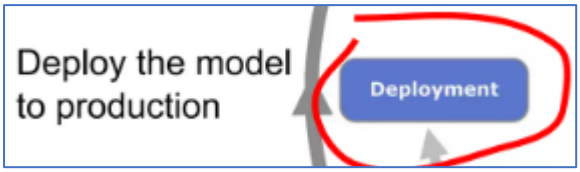
- Go back and adjust the goal
- Roll the model to more users/all users
- Stop working on the project

Evaluation + Deployment

Often happens together:

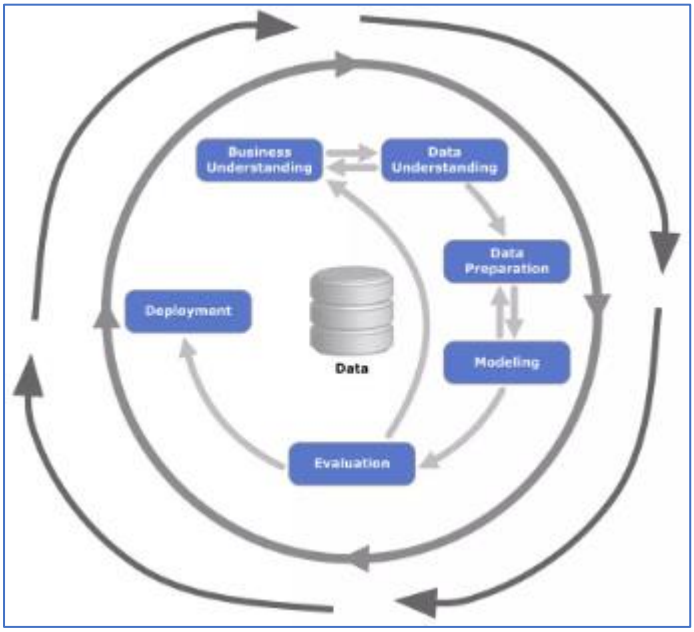
- Online evaluation: evaluation of live users
- It means: deploy the model, evaluate it

Deployment

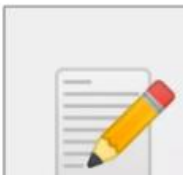


- Roll the model to all users
- Proper monitoring
- Ensuring the quality and maintainability

Iterate



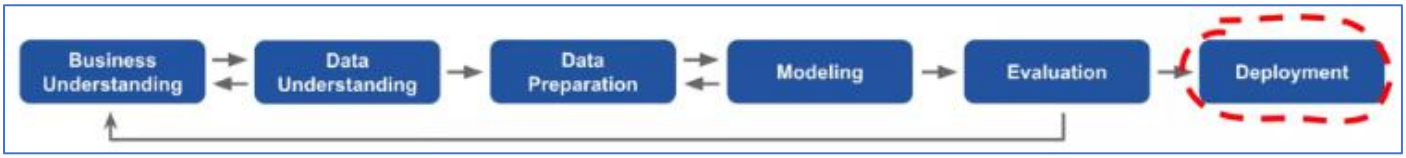
ML projects require many iterations!



Start simple
Learn from feedback
Improve

Summary

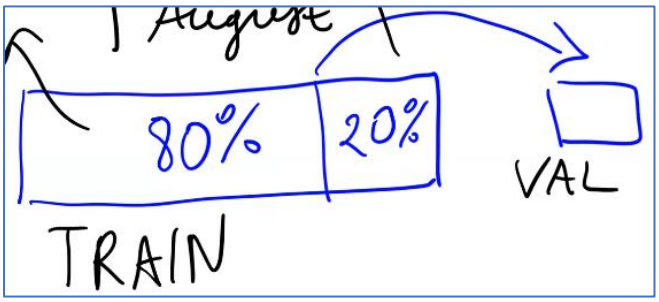
- Business understanding: define a measurable goal. Ask: do we need ML?
- Data understanding: do we have the data? Is it good?
- Data preparation: transform data into a table, so we can put it into ML
- Modelling: to select the best model, use the validation set
- Evaluation: validate that the goal is reached
- Deployment: roll out to production to all the users
- Iterate: start simple, learn from the feedback, improve

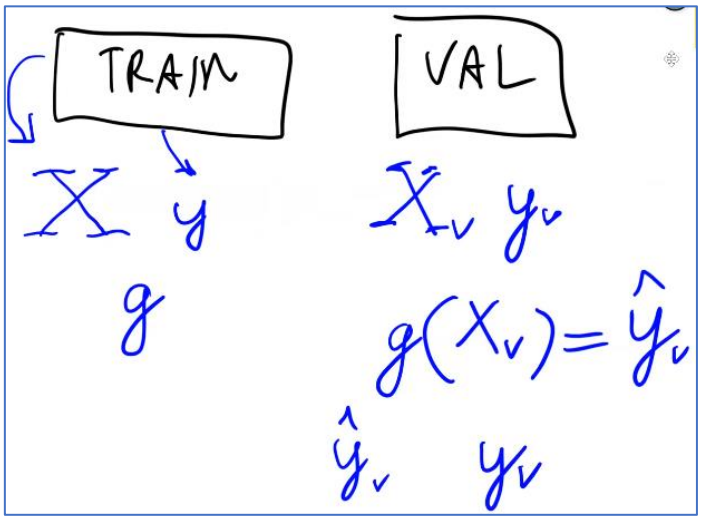


Modeling: Model Selection Process

You have many models like Logistic Regression, Decision Tree, Random Forest, Neural Network etc to solve your problem. But, which one does good and consistently requires you to evaluate each model

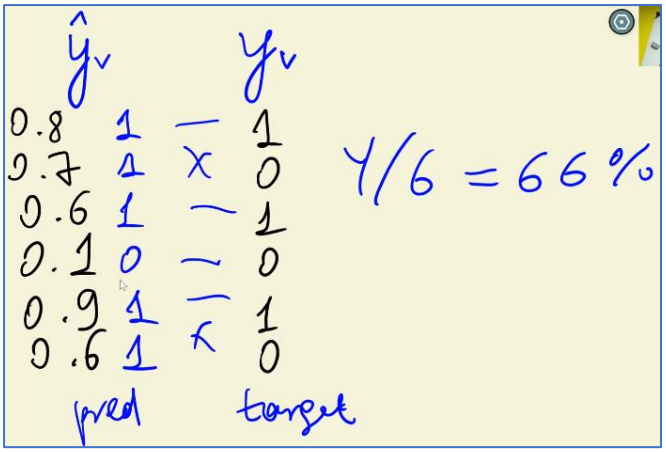
So, we have the Feature Matrix X, Target Vector y and the Model Function g. We split the X,y data into 2 parts - Training data and Validation data like 80%, 20%



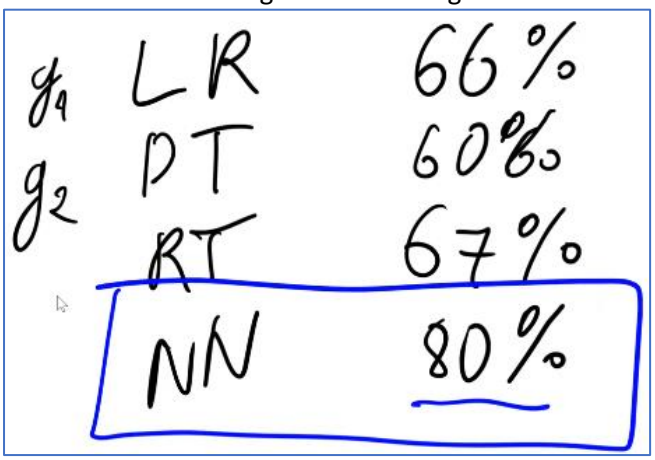


We train the model g on the Training dataset (X,y). Then, we make predictions y-hat-v_subscript by applying model g on X-v_subscript Validation data. And, compare the predicted values y-hat-v_subscript with the expected values y-v_subscript

In practice, for example the SPAM classification problem, we have the following Expected values and Predicted Probabilities. And, for Predicted values > 0.5, we treat it as 1, else 0. And when we compare y-hat-v_subscript and y-v_subscript, we see 4 out of 6 values match. So, accuracy is 4/6 = 66% for this model say Logistic Regression.



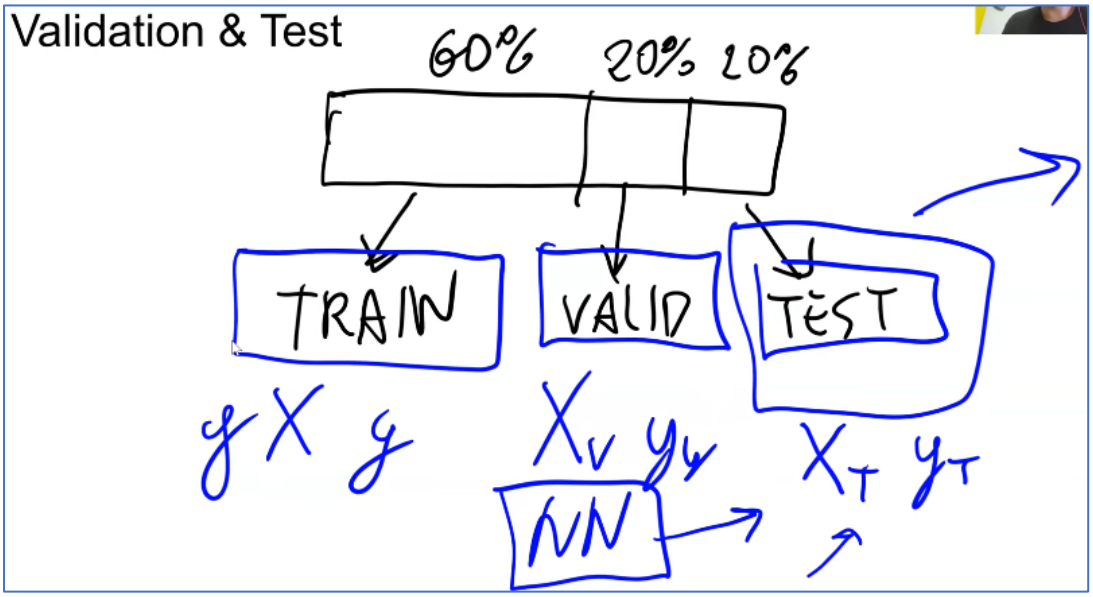
We repeat the process of training, making predictions, comparing predictions with validation, measuring accuracy for other models and get the following results.



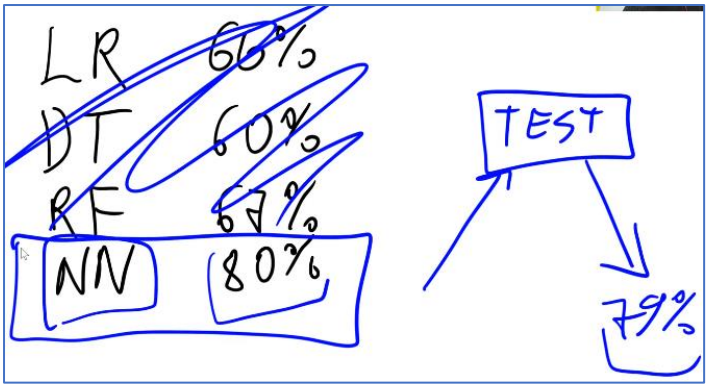
So, we see that Neural network has got highest score, so we would tend to select it.

But, because all the models are based on Probabilities, it may be possible that NN was plain lucky to have got the predictions correct with the given training data. So, to eliminate the chance of it being plain lucky, we divide original dataset (X,y) into 3 parts - Training (60%), Validation (20%), Testing (20%)

Training + Validation + Test method

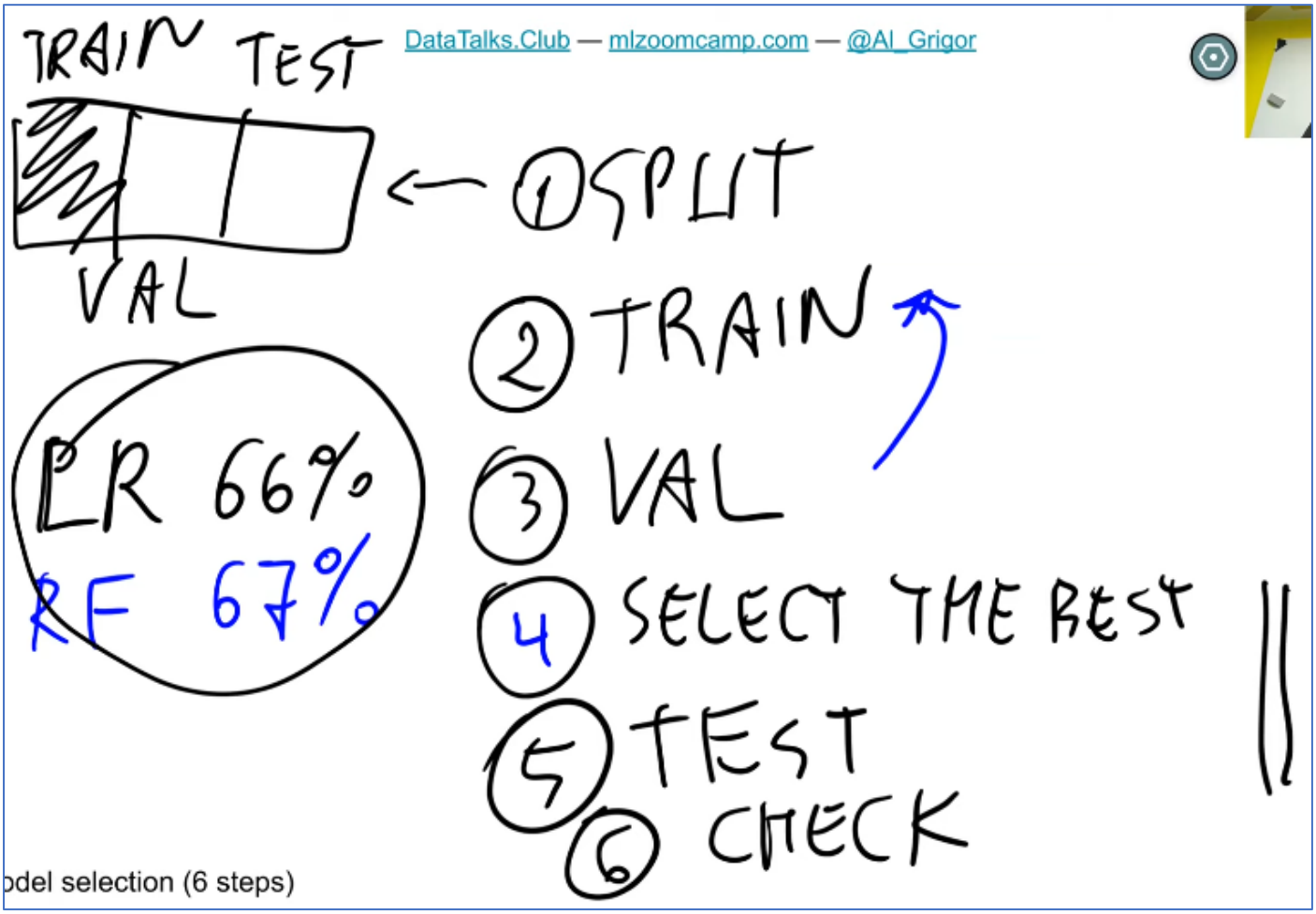


In this, we split the Original dataset (X,y) into 3 parts - Training (60%), Validation (20%), Testing (20%). We train different models g1, g2 etc on the training part, make predictions on X-v_subscript Validation data, compare each y-hat-v_subscript with y-v_subscript Validation data and score each model to get a score, s-v_subscript.



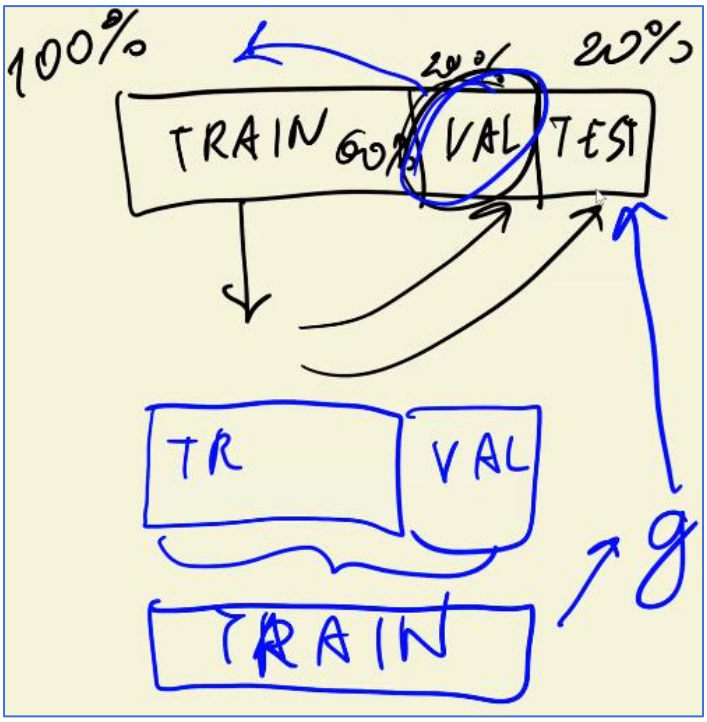
The model with highest score is fed with X-t_subscript Test data to make predictions y-hat-t_subscript which is compared with y-t_subscript Test data to generate a score, s-t_subscript. If s-t_subscript is close to s-v_subscript, then we can conclude that this model behaves quite well and can be finally selected ➔ MODEL SELECTION PROCESS

Model Selection 6 steps



1. Split Dataset into 3 parts - Training (60%), Validation (20%), Test (20%)
2. Train different models using Training data
3. Apply each model to Validation data, make predictions and calculate accuracy
4. Compare accuracies and select the best score model
5. To ensure best scoring model did not get plain lucky, apply that model to the Test data
6. Check the prediction accuracy of Test data with the Accuracy got with Validation data. If they are close enough, we have selected the model satisfactorily. Else, repeat Step 5,6 for the next scored model

Is the Validation Data wasted - Alternative approach?



When we split the data into training (60%), validation (20%), testing (20%), the validation data is just being used for checking each model and is kind of wasted.

So, to avoid wastage, after Model Selection Process is complete, the Training and Validation data can be combined to form a larger Training set.

The selected model can be trained again on the bigger set to make it better and checked against the Test data to see any change in accuracy/performance

Here are the steps for the alternative approach:

1. Split the original dataset into training, validation, and test sets with a ratio of 60%-20%-20%.
2. Train the initial models using the training dataset.
3. Apply the initial models to the validation dataset and evaluate their performance.
4. Select the best-performing model based on the validation results.
5. Combine the training and validation datasets to create a new combined dataset.
6. Retrain the selected model using the new combined dataset.
7. Apply the newly trained model to the test dataset to assess its performance on unseen data.

By training the model on a larger combined dataset, we can potentially capture more patterns and improve the model's ability to generalize to new data. The final evaluation on the test dataset provides a more reliable measure of the model's performance and gives us confidence in its ability to make accurate predictions.

It's important to note that the alternative approach may not always yield better results compared to the original model selection process. The effectiveness of this approach depends on the specific characteristics of the dataset and the performance of the initial models. Experimentation and careful evaluation are key to determine the most suitable approach for your machine learning task.

Development Environment Setup

It is preferred to develop ML projects and scripts on Linux OS with adequate hardware of RAM. So, if your PC has Windows OS (Win 10 or higher), then use Windows Subsystem for Linux (WSL) feature which allows you to install a Linux distro and run it within Windows without requiring to install a VM or dual-boot config. Within WSL's Linux, you can install Python, Docker and other dev tools that you need but this will depend on how much resources your PC has.

Alternatively, you can use Github Codespaces (free tier) which provides the entire dev environment on the cloud and you just need a browser to access it. You can use VSCode right in the browser or use the Desktop version to directly edit and debug your cloud-based code files

Machine Learning Development vs Traditional Software Development

In Traditional Software Development, when I write a piece of software, ideally it's a "write once run many" situation; the code is going to be distributed to user workstations or deployed to a server where it runs for a while, so the development environment is about fluent authoring, codebase management, planning for maintainability, and writing good tests so I can have confidence that I'm deploying good code. So, those environments look like VS Code - text editors with ample features for authoring and managing projects.

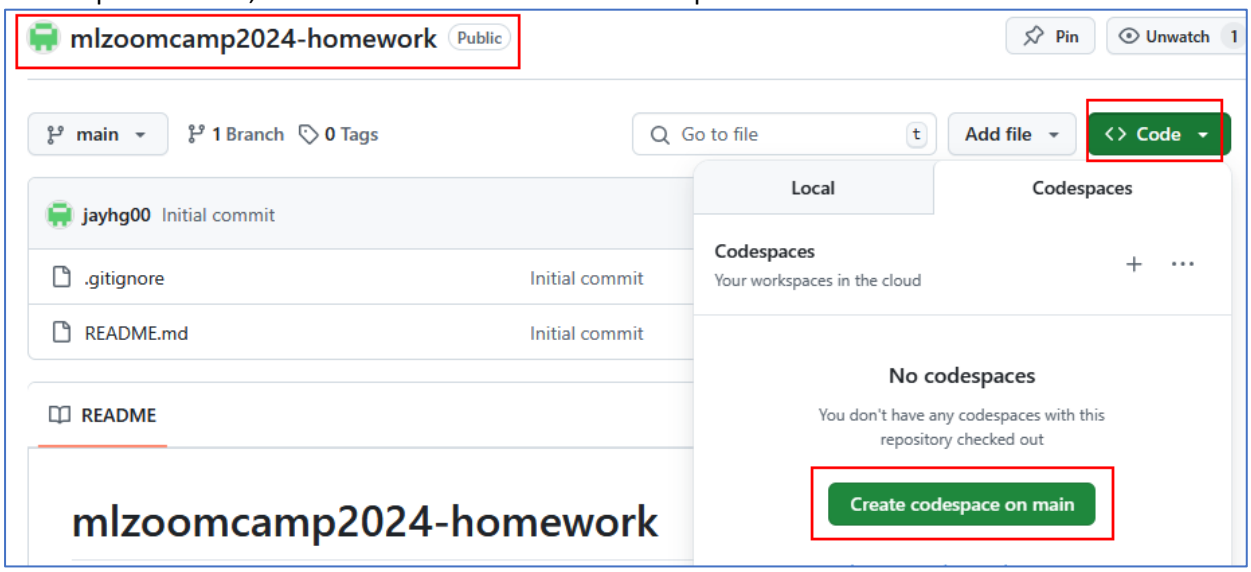
On the other hand, Machine Learning Development heavily uses Data Science which in itself is an exploratory, experimental subject. It's a "write many run once" situation - lots of experimentation, lots of need to debug in runtime before your model can be considered to be ready for deployment (deployment will be as an API to receive the request, apply the model on the request and send back the response). So, we use Notebooks for ML Development to run individual blocks in isolation with live debugging/rendering of nontextual outputs, lots of comments to explain why we are doing what etc. There are tools like Jupytext which can create your production-ready code for deployment from your Jupyter notebook

Setup in Github Codespaces

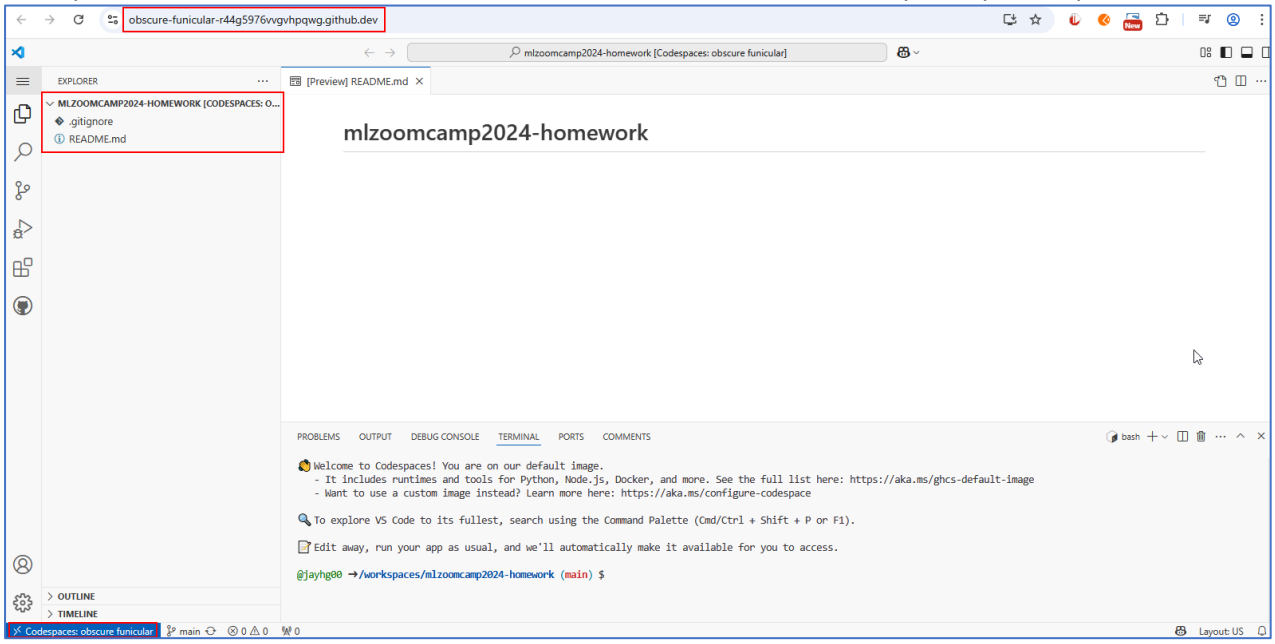
In Free Tier, you get free monthly limits of 15GB, 120 Core-hours. We will create the development environment with Python 3.11, Numpy, Pandas, Sci-kit learn, Seaborn, Jupyter notebook in Codespace

1. Login to personal Github account and Create new repo
   Set name,
   readme file,
   gitignore = Python

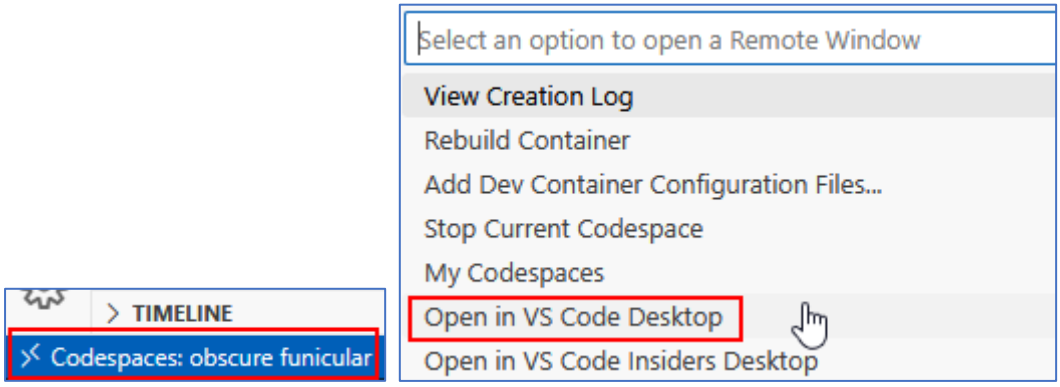
2. Once repo is created, click “Code” → “ click “Create codespace on main”



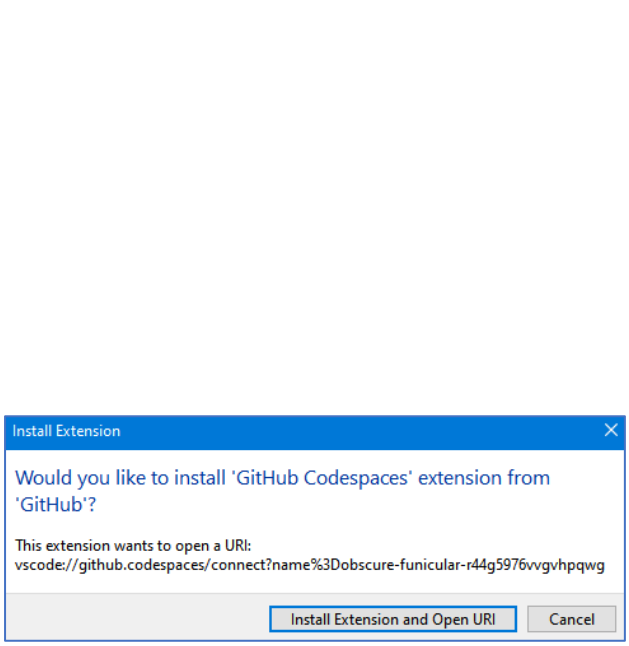
3. This opens a new browser tab with online VSCode IDE. The online VSCode opens up the repo folder structure

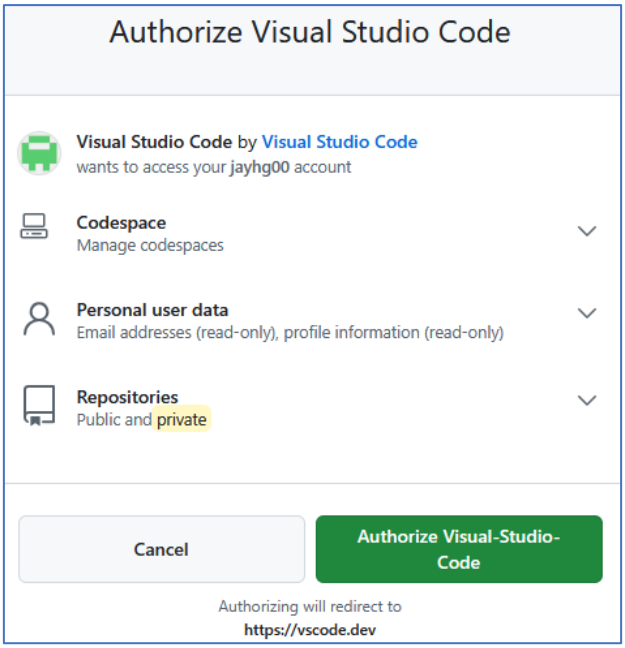


4. To open the remote repo directly in VSCode Desktop, click the Codespace name in bottom-left corner of the online VSCode. This will bring a command list and click “Open in VS Code Desktop”

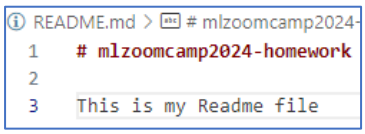


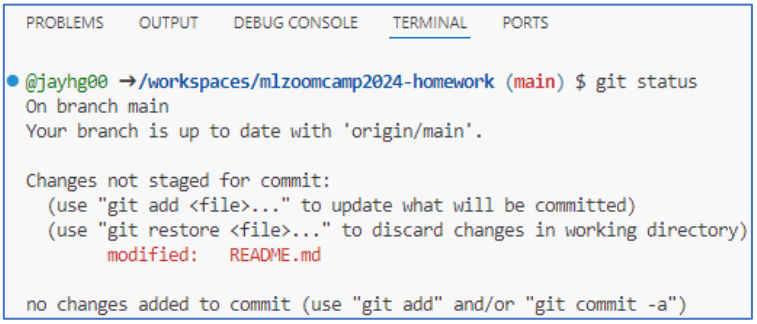
5. VSCode Desktop IDE opens. If this is the first time you are using Github Codespace in the desktop, you will be prompted to install extension “Github Codespaces”. Install it. And then authorize VSCode Desktop to your Github repos/profiles



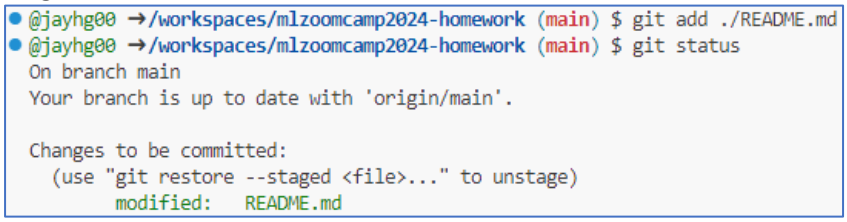


6. From this Desktop IDE, you can develop as if you are doing it locally and then use the terminal to commit and push your changes to Git
   a. Modified Readme.md file





   b. Staged the file



c. Committed the file

```
@jayhg00 ➜/workspaces/mlzoomcamp2024-homework (main) $ git commit -am 'Readme update'
[main 3d0ec35] Readme update
 1 file changed, 3 insertions(+), 1 deletion(-)
@jayhg00 ➜/workspaces/mlzoomcamp2024-homework (main) $ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

d. Push to Github

```
@jayhg00 ➜/workspaces/mlzoomcamp2024-homework (main) $ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 340 bytes | 340.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/jayhg00/mlzoomcamp2024-homework
   fb82914..3d0ec35  main -> main
```

```
@jayhg00 ➜/workspaces/mlzoomcamp2024-homework (main) $ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

7. To make the command prompt text short, set the PS1 environment variable. We set it to just '$ '

```
@jayhg00 ➜/workspaces/mlzoomcamp2024-homework (main) $ PS1="$ "
$
```

8. Install the ML libraries that we need using Python's Package Manager, pip. Run following command-
pip install jupyter numpy pandas scikit-learn seaborn

```
$ pip install jupyter numpy pandas scikit-learn seaborn
Collecting jupyter
  Downloading jupyter-1.1.1-py2.py3-none-any.whl.metadata (2.0 kB)
Requirement already satisfied: numpy in /home/codespace/.local/lib/python3.12/site-packages (2.2.4)
Requirement already satisfied: pandas in /home/codespace/.local/lib/python3.12/site-packages (2.2.3)
Requirement already satisfied: scikit-learn in /home/codespace/.local/lib/python3.12/site-packages (1.6.1)
Requirement already satisfied: seaborn in /home/codespace/.local/lib/python3.12/site-packages (0.13.2)
```

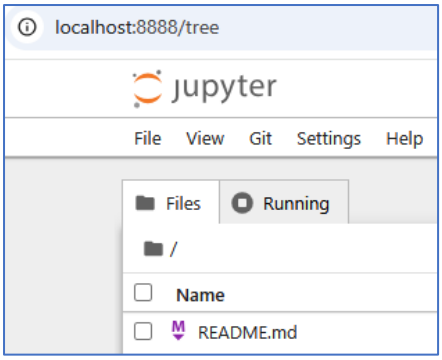
9. To run a Jupyter notebook remotely, run the following-
$ jupyter notebook

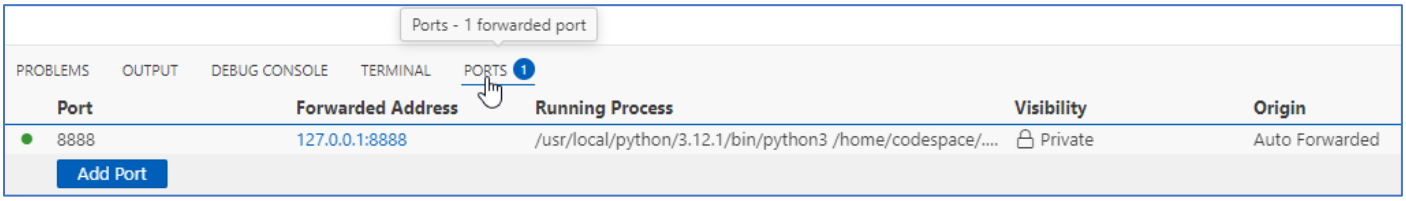
```
$ jupyter notebook
[I 2025-03-28 06:32:33.263 ServerApp] jupyter_lsp | extension was successfully linked.
[I 2025-03-28 06:32:33.267 ServerApp] jupyter_server_mathjax | extension was successfu
[I 2025-03-28 06:32:33.270 ServerApp] jupyter_server_terminals | extension was success
[W 2025-03-28 06:32:33.272 LabApp] 'allow_origin' has moved from NotebookApp to Server
```

When Jupyter environment is initialized on the server, the following URLs are provided for accessing it

```
To access the server, open this file in a browser:
    file:///home/codespace/.local/share/jupyter/runtime/jpserver-16394-open.html
Or copy and paste one of these URLs:
    http://localhost:8888/tree?token=ea80cdd5b0e0576a24ff1f4174b0753d7148cb7e9c87bdcf
    http://127.0.0.1:8888/tree?token=ea80cdd5b0e0576a24ff1f4174b0753d7148cb7e9c87bdcf
```

10. Navigate to the localhost full URL in browser. Jupyter Notebook environment loads with our repo files. This is done by using Port forwarding. In VSCode Desktop beside Terminal tab, you see Ports tab which lists all the forwarded ports





11. In this online Jupyter Notebook workspace, you can create new notebooks/import notebook run them.

OR

Local VSCode Setup

1. Install Python extension,
2. Within VSCode terminal, install Python libraries using pip install command
   >> pip install numpy pandas scikit-learn jupyter seaborn
3. Install Jupyter notebook VSCode extension to run .ipynb files within VSCode
4. Download the .ipynb files from the course's Github repo and run them locally through VSCode desktop

Intro to Numpy

Numpy provides support for large, multi-dimensional arrays and matrices, along with a wide range of mathematical functions to operate on these arrays. Key feature of NumPy is its ability to perform vectorized operations, which allows for faster and more efficient computations. Instead of looping over individual elements of an array, we can perform operations on the entire array at once.

You can do all the operations on Numpy Array that you can do with Python lists like accessing by negative indexing, slicing,

Refer to Jupyter notebook numpy.ipynb in this folder for theory/code

Creating arrays

```
import numpy as np

my_array = np.array([1, 2, 3, 4, 5])
```
→ array([1, 2, 3, 4, 5])

With specific datatype

```
my_float_array = np.array([1.1, 2.2, 3.3, 4.4, 5.5], dtype=float)
```
→ array([1.1, 2.2, 3.3, 4.4, 5.5])

Array from a range of numbers similar to List from range

```
my_range_array = np.arange(1, 10, 2)
```
→ array([1, 3, 5, 7, 9])

```
# argument is the size of the array
# result is an array with 5 zeros
np.zeros(5)
# Output:
# array([0., 0., 0., 0., 0.])

np.ones(10)
# Output:
# array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

# first argument is the size of the array
# second argument is the element you want to fill the array with
np.full(10, 2.5)
# Output:
# array([2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5])
```

```
# create an array with the range from 0 to <argument>
np.arange(10)
# Output:
# array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(3, 10)
# Output:
# array([3, 4, 5, 6, 7, 8, 9])

# linspace creates an array filled with numbers between first argumen
np.linspace(0, 1, 11)
# Output:
# array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Converting a Python List

Use np.array(list) method only

```
import numpy as np

my_list = [1, 2, 3, 5, 7, 12]
my_array = np.array(my_list)

# Output:
# array([ 1,  2,  3,  5,  7, 12])
```

Accessing array elements

array_name[index]

```
my_array = np.array([1, 2, 3, 4, 5])
```

```
second_element = my_array[1]
```
→ 2

```
last_element = my_array[-1]
```
→ 5

Slicing - array_name[start_index:end_index:step]

```
subset_array = my_array[1:4]
```
→ [2, 3, 4],

```
subset_array = my_array[2:]
```
→ [3, 4, 5]

Multi-dimensional array

Pass in nested list or tuple to np.array()

```
# Create a 2-dimensional array
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

→

```
[[1 2 3]
 [4 5 6]]
```

Element-wise operations

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Element-wise addition
result = arr1 + arr2
print(result)
# Output:
# [[ 6  8]
# [10 12]]

# Element-wise multiplication
result = arr1 * arr2
print(result)
# Output:
# [[ 5 12]
#  [21 32]]
```

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Calculate the sum of all elements
sum_val = np.sum(arr)
print(sum_val)
# Output:
# 21

# Calculate the mean of each column
mean_val = np.mean(arr, axis=0)
print(mean_val)
# Output:
# [2.5 3.5 4.5]
```

Zeros, Ones in 2d array

```
# creates an array with 5 rows and 2 columns, filled with zeros
np.zeros((5, 2))
# Output:
#array([[0., 0.],
#       [0., 0.],
#       [0., 0.],
#       [0., 0.],
#       [0., 0.]])

# creates an array from a python list
# here we have a list of lists
np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
# Output:
# array([[1, 2, 3],
#       [4, 5, 6],
#       [7, 8, 9]])
```

Accessing elements in 2d array

comma separated indexes in []

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Accessing a single element
element = arr[0, 2]
print(element)
# Output: 3
```

Access single row (: for all columns)

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Accessing a single row
row = arr[1, :]
print(row)
# Output: [4 5 6]
```

Access specific of rows (List of rowindex, : for all columns)

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing multiple rows
rows = arr[[0, 2], :]
print(rows)
# Output:
# [[1 2 3]
#  [7 8 9]]
```

Replace a row

Set the row to another array

```
n = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# outputs the first row
n[0]
# Output:
# array([ 1, 20,  3])

n[0] = [12, 13, 14]
n
# Output:
# array([[12, 13, 14],
#        [ 4,  5,  6],
#        [ 7,  8,  9]])
```

Access single column (: for all rows)

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

# Accessing a single column
column = arr[:, 1]
print(column)
# Output: [2 5]
```

Access specific columns (: for all rows, list of index)

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing multiple columns
columns = arr[:, [0, 2]]
print(columns)
# Output:
# [[1 3]
#  [4 6]
#  [7 9]]
```

Replace a column
Set column to another array

```
n = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# outputs the first column
n[:, 0]
# Output:
# array([1, 4, 7])

n[:, 0] = [12, 13, 14]
n
# Output:
# array([[12,  2,  3],
#        [13,  5,  6],
#        [14,  8,  9]])
```

Randomly generated arrays

Useful for scientific and math applications, testing and simulation purposes. Use numpy.random module

```python
import numpy as np

# Generate a 1-dimensional array of 5 random integers between 0 and 9
random_integers = np.random.randint(10, size=5)
print(random_integers)
# Output: [2 5 7 1 8]

# Generate a 2-dimensional array of shape (3, 4)
# with random floats between 0 and 1
random_floats = np.random.random((3, 4))
print(random_floats)
# Output:
# [[0.0863851  0.83574087 0.79192621 0.85248822]
#  [0.14040051 0.5714931  0.7586195  0.48544792]
#  [0.4304003  0.76688989 0.68447497 0.54361942]]

# Generate a 3-dimensional array of shape (2, 3, 2)
# with random values from a standard normal distribution
random_normal = np.random.normal(size=(2, 3, 2))
print(random_normal)
# Output:
# [[[-0.27013393  0.54416022]
#   [ 0.02537238 -0.78380969]
#   [-1.25909646 -1.04630766]]
#
#  [[ 0.34262622 -0.66770036]
#   [-0.35426751  0.00569635]
#   [-0.05665257  0.02068191]]]
```

```
# generates a 2-dimensional array of size 5 rows and 2 columns
# with random numbers between 0 and 1
# rand samples from standard uniform distribution
np.random.rand(5, 2)
# Output:
# array([[0.83575882, 0.03277884],
#        [0.78785763, 0.34340225],
#        [0.79212789, 0.75564912],
#        [0.78937584, 0.4326158 ],
#        [0.90909093, 0.82098053]])

# when you set the random seed it's possible to reproduce the
# same "random" values
np.random.seed(2)
np.random.rand(5, 2)
# Output:
# array([[0.4359949 , 0.02592623],
#        [0.54966248, 0.43532239],
#        [0.4203678 , 0.33033482],
#        [0.20464863, 0.61927097],
#        [0.29965467, 0.26682728]])

# randn samples from standard normal distribution
np.random.seed(2)
np.random.randn(5, 2)
# Output:
# array([[-0.41675785, -0.05626683],
#        [-2.1361961 ,  1.64027081],
#        [-1.79343559, -0.84174737],
#        [ 0.50288142, -1.24528809],
#        [-1.05795222, -0.90900761]])

# creates random numbers between 0 and 100
np.random.seed(2)
100 * np.random.rand(5, 2)
# Output:
# array([[43.59949021,  2.59262318],
#        [54.96624779, 43.53223926],
#        [42.03678021, 33.0334821 ],
#        [20.4648634 , 61.92709664],
#        [29.96546737, 26.68272751]])

# creates an array of random integer numbers
np.random.seed(2)
np.random.randint(low=0, high=100, size=(5, 2))
# Output:
# array([[40, 15],
#        [72, 22],
#        [43, 82],
#        [75,  7],
#        [34, 49]])
```

Element-wise operations

With a list, you will need to iterate through each element to do the operation. With Numpy array, you can simply apply the operation to the array and Numpy internally applies to each element in an efficient manner

```python
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result = arr1 + arr2
print(result)
# Output: [5 7 9]

# Element-wise subtraction
result = arr1 - arr2
print(result)
# Output: [-3 -3 -3]
```

```python
a = np.arange(5)
a
# Output:
# array([0, 1, 2, 3, 4])

# adds 1 to every element in the array
# be careful, you cannot do this with a normal python list
a + 1
# Output:
# array([1, 2, 3, 4, 5])

a * 2
# Output:
# array([0, 2, 4, 6, 8])

a * 100
# Output:
# array([  0, 100, 200, 300, 400])

a / 100
# Output:
# array([0.  , 0.01, 0.02, 0.03, 0.04])

(10 + (a * 2)) ** 2
# Output:
# array([100, 144, 196, 256, 324])

b = (10 + (a * 2)) ** 2 / 100
b
# Output:
# array([1.  , 1.44, 1.96, 2.56, 3.24])

# adds element-wise both arrays
a + b
# Output:
# array([1.  , 2.44, 3.96, 5.56, 7.24])
```

Comparison operations

Allows you to compare elements of arrays and obtain Boolean results. Useful for tasks such as filtering or conditional assignment.

```python
arr = np.array([1, 2, 3, 4, 5])

# Compare elements with 3
result = arr > 3
print(result)
# Output: [False False False  True  True]
```

```python
arr1 = np.array([1, 2, 3])
arr2 = np.array([3, 2, 1])

# Compare elements of arr1 and arr2
result = arr1 == arr2
print(result)
# Output: [False  True False]
```

```python
arr = np.array([1, 2, 3, 4, 5])

# Compare elements that are greater than 2 and less than 5
result = (arr > 2) & (arr < 5)
print(result)
# Output: [False False  True  True False]
```

Filtering elements using comparison

```
a = np.arange(5)
b = (10 + (a * 2)) ** 2 / 100
# compare numbers element-wise
a >= 2
# Output:
# array([False, False,  True,  True,  True])

a > b
# Output:
# array([False, False,  True,  True,  True])

# checks which elements of a are greater than b
# a > b -> returns an boolean array
# a[a > b] returns the elements where the boolean array is true
# that are the elements 2, 3, and 4
a[a > b]
# Output:
# array([2, 3, 4])

a[2], a[3], a[4]
# Output:
(2, 3, 4)
```

Summarizing (Aggregate) functions

These return a single value

```
# there are some operations that instead of returning a new array,
# it returns a single number
# e.g. min() returns the smallest number
a.min()
# Output: 0

a.max()
# Output: 4

a.sum()
# Output: 10

a.mean()
# Output: 2

# standard deviation
a.std()
# Output: 1.4142135623730951

n = np.array([
   [12, 13,  0],
    [ 4,  5,  1],
    [ 7,  8,  2]
])

# this also works for 2-dimensional arrays
n.sum()
# Output: 52

n.min()
# Output: 0
```

Linear Algebra Refresher

Vector is a 1-d array (1-column array) denoted in lowercase (a)
Matrix is a 2-d array denoted in uppercase (A)

Though the following operations are done under the hood by various ML algos/libs, it is good to know the inner mathematical concepts

Simple Vector Operations

Scalar Vector Product (Scalar Multiplication)

Multiply scalar (single) value to each element of vector. Used to scale vectors, change their direction, or perform transformations in vector spaces. Heavily used in linear combinations, linear transformations, and eigenvector calculations

$$2 \cdot \begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \\ 10 \\ 12 \end{bmatrix}$$

→

```
u = np.array([2, 4, 5, 6])
2 * u
# Output:  array([4, 8, 10, 12])
```

Vector addition

adding two vectors together to obtain a new vector → Element-wise addition

$$\begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 2+1 \\ 4+0 \\ 5+0 \\ 6+2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 8 \end{bmatrix}$$

→

```
u = np.array([2, 4, 5, 6])
v = np.array([1, 0, 0, 2])
u + v
# Output:  array([3, 4, 5, 8])
```

Vector Vector Multiplication (Dot Product of Vectors, Scalar Product)

Multiply the corresponding components of two vectors and sum them up. Returns a single value hence called Scalar Product also

$$U \cdot V = U_1V_1 + U_2V_2 + U_3V_3 + \ldots + U_nV_n$$

$$\begin{bmatrix} 2 \\ 4 \\ 5 \\ 6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} = (2*1) + (4*0) + (5*0) + (6*2) = 2 + 12 = 14$$

$$v^T = [2\ 4\ 5\ 6] \qquad u = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 2 \end{bmatrix} \qquad v^T u = \sum_{i=1}^{n} u_i\, v_i$$

Scalar Product provides information about the similarity or alignment of two vectors. For example, if the dot product of two vectors is zero, it indicates that the vectors are perpendicular or orthogonal to each other. Conversely, a positive dot product suggests that the vectors have a similar direction, while a negative dot product indicates opposite directions. It allows us to calculate the angle between vectors, determine vector projections, and perform vector comparisons. In machine learning, the dot product is particularly useful in computing similarity measures,

such as cosine similarity, which is commonly used in recommendation systems and natural language processing tasks.

Implementation Logic-

```python
import numpy as np

def vector_vector_multiplication(u, v):
    assert u.shape[0] == v.shape[0]

    n = u.shape[0]

    result = 0.0

    for i in range(n):
        result = result + u[i] * v[i]

    return result

u = np.array([2, 4, 5, 6])
v = np.array([1, 0, 0, 2])
vector_vector_multiplication(u, v)
# Output: 14.0

# dot product is already implemented in numpy
u.dot(v)
# Output: 14.0
```

u.shape - Returns tuple (no of rows, no of columns).
u.shape(0) - To get 1st element of tuple i.e. no of rows
Assert that both the vectors, u and v, need to have the same no of rows. If false, script will raise AssertionError and stop

Iterate through both u and v, multiply each element and add those products

However, numpy provides a dot() function that abstracts this logic

Matrix Vector Multiplication

Multiply matrix (m x n) with vector (n x 1) to produce new vector (m x 1)

It is used to perform transformations, solve systems of linear equations, and represent linear mappings between vector spaces. In machine learning, matrix vector multiplication is fundamental in performing linear regression, applying weight matrices to input data, and transforming features in neural networks.

Mathematically, if we have a matrix U and a vector v, the matrix vector multiplication is denoted as Uv and is calculated as:

$$Uv = (U_{11}v_1 + U_{12}v_2 + \ldots + U_{1n}v_n,\ U_{21}v_1 + U_{22}v_2 + \ldots + U_{2n}v_n,\ \ldots,\ U_{m1}$$

Here, U_{11}, U_{12}, ..., U_{mn} are the individual elements of the matrix U, and v_1, v_2, ..., v_n are the components of the vector v.

$$U \cdot v = Uv$$

$$\begin{matrix} u_0 \longrightarrow \\ u_1 \longrightarrow \\ u_2 \longrightarrow \end{matrix} \begin{bmatrix} 2 & 4 & 5 & 6 \\ 1 & 2 & 1 & 2 \\ 3 & 1 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0.5 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} u_0^T v \\ u_1^T v \\ u_2^T v \end{bmatrix}$$

Matrix vector multiplication is vector-vector multiplication for each row of matrix U with vector v

```python
import numpy as np

def matrix_vector_multiplication(U, v):
    assert U.shape[1] == v.shape[0]

    num_rows = U.shape[0]

    result = np.zeros(num_rows)

    for i in range(num_rows):
        result[i] = vector_vector_multiplication(U[i], v)

    return result

U = np.array([
    [2, 4, 5, 6],
    [1, 2, 1, 2],
    [3, 1, 2, 1]
])

v = np.array([1, 0, 0, 2])

matrix_vector_multiplication(U, v)
# Output: array([14.,  5.,  5.])

# dot product between vector and matrix is already implemented in numpy
U.dot(v)
# Output: array([14,  5,  5])
```

Again, you can simply use Numpy.dot() function to get this U.v

Matrix Matrix multiplication

Multiply Matrix (m x n) with matrix (n x p) to give matrix (m x p)
Multiply each row of the first matrix by each column of the second matrix, and then sum up the results.

Mathematically, if we have two matrices, A and B, the matrix matrix multiplication is denoted as AB and is calculated as:

$AB = (A_{11}B_{11} + A_{12}B_{21} + \ldots + A_{1n}B_{n1}, A_{11}B_{12} + A_{12}B_{22} + \ldots + A_{1n}B_{n2}, \ldots, A_{m1}B_{11} + A_{m2}B_{21} + \ldots + A_{mn}B_{n1};$

$A_{11}B_{12} + A_{12}B_{22} + \ldots + A_{1n}B_{n2}, A_{11}B_{12} + A_{12}B_{22} + \ldots + A_{1n}B_{n2}, \ldots, A_{m1}B_1 \rightarrow A_{m2}B_{22} + \ldots + A_{mn}B_{n2};$

$\ldots$

$A_{11}B_{1m} + A_{12}B_{2m} + \ldots + A_{1n}B_{nm}, A_{11}B_{1m} + A_{12}B_{2m} + \ldots + A_{1n}B_{nm}, \ldots, A_{m1}B_{1m} + A_{m2}B_{2m} + \ldots + A_{mn}B_{mm})$

Here, $A_{11}, A_{12}, \ldots, A_{mn}$ and $B_{11}, B_{21}, \ldots, B_{nm}$ are the individual elements of matrices A and B, respectively.

$$U \quad \times \quad V \quad = \quad UV$$

$$\begin{bmatrix} 2 & 4 & 5 & 6 \\ 1 & 2 & 1 & 2 \\ 3 & 1 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 2 \\ 0 & 0.5 & 1 \\ 0 & 2 & 1 \\ 2 & 1 & 0 \end{bmatrix} = \begin{bmatrix} | & | & | \\ Uv_0 & Uv_1 & Uv_{02} \\ | & | & | \end{bmatrix}$$

$$\uparrow \quad \uparrow \quad \uparrow$$
$$v_0 \quad v_1 \quad v_2$$

Matrix Matrix multiplication is Matrix-Vector Multiplication of Uv0, Uv1, Uv2

Applied in linear transformations, optimizing systems of equations, and representing complex mathematical operations. In machine learning, matrix matrix multiplication is used in tasks such as matrix factorization, neural network training, and model optimization.

```
import numpy as np

def matrix_matrix_multiplication(U, V):
    assert U.shape[1] == V.shape[0]

    num_rows = U.shape[0]
    num_columns = V.shape[1]

    result = np.zeros((num_rows, num_columns))

    for i in range(num_columns):
        vi = V[:, i]
        Uvi = matrix_vector_multiplication(U, vi)
        result[:, i] = Uvi

    return result

U = np.array([
    [2, 4, 5, 6],
    [1, 2, 1, 2],
    [3, 1, 2, 1]
])

V = np.array([
    [1, 1, 2],
    [0, 0.5, 1],
    [0, 3, 1],
    [2, 1, 0]
])

matrix_matrix_multiplication(U, V)
# Output:
#array([[14. , 25. , 13. ],
#       [ 5. ,  7. ,  5. ],
#       [ 5. , 10.5,  9. ]])

# dot product between matrix and matrix is already implemented in numpy
U.dot(V)
# Output:
#array([[14. , 25. , 13. ],
#       [ 5. ,  7. ,  5. ],
#       [ 5. , 10.5,  9. ]])
```

Again, you can simply use Numpy.dot() function to get this U.V

Identity Matrix I

Special type of square matrix in linear algebra. It is denoted as I and has ones along its main diagonal (from the top-left to the bottom-right) and zeros in all other positions.
It behaves like the number one in matrix operations. When the identity matrix is multiplied with another matrix U, the result is U itself. Mathematically, this can be expressed as:
I * U = U

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Identity matrix of size 4 x 4

Similarly, when matrix U is multiplied by the identity matrix, the result is also U:

U * I = U

It serves as the neutral element for matrix multiplication, similar to how the number one acts as the neutral element for multiplication of real numbers. The identity matrix also plays a crucial role in defining matrix inverses, solving systems of linear equations, and performing transformations.
In machine learning, the identity matrix is often used as the initial value for weight matrices in neural networks. This ensures that the initial weights do not affect the input data during the first round of computations. The use of identity matrices in neural networks helps prevent over-fitting and facilitates better convergence during the training process.

In numpy, you use np.eye() function to create Identity matrix

```
import numpy as np

np.eye(3)
# Output:
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])

V = np.array([
    [1, 1, 2],
    [0, 0.5, 1],
    [0, 3, 1],
    [2, 1, 0]
])

V.dot(I)
# Output:
# array([[1. , 1. , 2. ],
#        [0. , 0.5, 1. ],
#        [0. , 3. , 1. ],
#        [2. , 1. , 0. ]])

V.dot(I) == V
# Output:
# array([[ True,  True,  True],
#        [ True,  True,  True],
#        [ True,  True,  True],
#        [ True,  True,  True]])
```

Inverse Matrix

Denoted as U^{-1} and represents the matrix that, when multiplied with the original matrix U, yields the identity matrix I.

$U^{-1}.U = I$

To calculate the inverse of a matrix U, we need to ensure that U is a square matrix and that it is invertible (i.e., its determinant is non-zero). Here is the general formula for finding the inverse:

$U^{-1} = (1/|U|) * adj(U)$

In this formula, $|U|$ represents the determinant of matrix U, and adj(U) denotes the adjugate of matrix U. The adjugate of a matrix is the transpose of its cofactor matrix.

Not all matrices are invertible. If a matrix is not invertible (i.e., it has a determinant of zero), it is called a singular matrix. Singular matrices have zero as an eigenvalue and cannot be inverted.

In machine learning, the inverse of a matrix is often used in optimization algorithms and data transformations.

In Numpy, we have function np.linalg.inv(U)

```
import numpy as np

# only squared matrices has an inverse matrix
V = np.array([
    [1, 1, 2],
    [0, 0.5, 1],
    [0, 3, 1]
])

# there is a function in numpy available that returns the inverse
V_inv = np.linalg.inv(V)
V_inv
# Output:
# array([[ 1. , -2. ,  0. ],
#        [ 0. , -0.4,  0.4],
#        [ 0. ,  1.2, -0.2]])

# just to check that V_inv.dot(V) == I
Vs_inv.dot(Vs)
# Output:
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])
```

Intro to Pandas

Pandas library is to represent data in tabular form and perform analysis. It provides data structures and functions that are essential for data analysis and preprocessing tasks. With Pandas, you can easily load, manipulate, and analyze structured data.

One of the key data structures in Pandas is the **DataFrame**. It is a two-dimensional table-like structure that allows you to store and manipulate data in a row-column format. You can think of it as a spreadsheet or a SQL table.

Another data structure in Pandas is **Series**, which is a one-dimensional labeled array. It is similar to a column in a DataFrame and can be used to store and manipulate a single variable. Series are particularly useful when you need to perform operations on a specific column or extract a subset of data from a DataFrame.

Pandas internally uses many Numpy structures and functions. So when you import Pandas, you also need to import Numpy

DataFrames

It is basically a table. Consider “data” to be a list of list of car info

Create Dataframes from List of Lists

import numpy as np
import pandas as pd

```
data = [
    ['Nissan', 'Stanza', 1991, 138, 4, 'MANUAL', 'sedan', 2000],
    ['Hyundai', 'Sonata', 2017, None, 4, 'AUTOMATIC', 'Sedan', 27150],
    ['Lotus', 'Elise', 2010, 218, 4, 'MANUAL', 'convertible', 54990],
    ['GMC', 'Acadia',  2017, 194, 4, 'AUTOMATIC', '4dr SUV', 34450],
    ['Nissan', 'Frontier', 2017, 261, 6, 'MANUAL', 'Pickup', 32340],
]
```

df = pd.DataFrame(data)
print(df)

Output-

	0	1	2	3	4	5	6	7
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Here, since we did not provide column names, they are named “0-n”. Also, the rows are named “0-m” (we can also rename rows through indexes)

Providing column names-

```
columns = [
    'Make', 'Model', 'Year', 'Engine HP', 'Engine Cylinders',
    'Transmission Type', 'Vehicle_Style', 'MSRP'
]
```

df = pd.DataFrame(data, columns = columns)
print(df)

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Create Dataframes from List of Dictionaries

In a dictionary we explicitly specify the value for each column. That means a dictionary is a key-value structure, where the keys are the column names and the values are the car specific value for that key.

```
data = [
    {
        "Make": "Nissan",
        "Model": "Stanza",
        "Year": 1991,
        "Engine HP": 138.0,
        "Engine Cylinders": 4,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "sedan",
        "MSRP": 2000
    },
    {
        "Make": "Hyundai",
        "Model": "Sonata",
        "Year": 2017,
        "Engine HP": None,
        "Engine Cylinders": 4,
        "Transmission Type": "AUTOMATIC",
        "Vehicle_Style": "Sedan",
        "MSRP": 27150
    },
    {
        "Make": "Lotus",
        "Model": "Elise",
        "Year": 2010,
        "Engine HP": 218.0,
        "Engine Cylinders": 4,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "convertible",
        "MSRP": 54990
    },
    {
        "Make": "GMC",
        "Model": "Acadia",
        "Year": 2017,
        "Engine HP": 194.0,
        "Engine Cylinders": 4,
        "Transmission Type": "AUTOMATIC",
        "Vehicle_Style": "4dr SUV",
        "MSRP": 34450
    },
    {
        "Make": "Nissan",
        "Model": "Frontier",
        "Year": 2017,
        "Engine HP": 261.0,
        "Engine Cylinders": 6,
        "Transmission Type": "MANUAL",
        "Vehicle_Style": "Pickup",
        "MSRP": 32340
    }
]
```

```
df = pd.DataFrame(data)
print(df)
```

Output-

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

After loading a DataFrame from csv file or from sql query the first thing to do is to look at the first rows to get a fast overview about the data.

df.head() ➔ Returns first 5 rows
df.head(n=2) → Returns first 2 rows

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150

Pandas Series

It is a one-dimensional labeled array that can hold data of any type. It is similar to a column in a DataFrame and can be thought of as a single variable.

When you access a column in Dataframe, it returns a Series

Accessing column in Dataframe

a. Dot notation (works only when column does not contain whitespace)

```
df.Make
# Output:
# 0     Nissan
# 1    Hyundai
# 2      Lotus
# 3        GMC
# 4     Nissan
# Name: Make, dtype: object
```

b. Bracket notation (works for any column)

```
df['Make']
# Output:
# 0     Nissan
# 1    Hyundai
# 2      Lotus
# 3        GMC
# 4     Nissan
# Name: Make, dtype: object
```

Create a new subset of the DataFrame with a selection of columns using the bracket notation.

```
df[['Make', 'Model', 'MSRP']]
```

	Make	Model	MSRP
0	Nissan	Stanza	2000
1	Hyundai	Sonata	27150
2	Lotus	Elise	54990
3	GMC	Acadia	34450
4	Nissan	Frontier	32340

Creating a Series

Create a Series by passing a list or array of values to the pd.Series() function

```
make_series = pd.Series(df['Make'])
print(make_series)
```

Output-

```
0     Nissan
1    Hyundai
2      Lotus
3        GMC
4     Nissan
Name: Make, dtype: object
```

Add columns to Dataframe

Assign new Series/List to new column name

```
df['id'] = [10, 20, 30, 40, 50]
print(df)
```

Output-

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP	id
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000	10
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150	20
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990	30
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450	40
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340	50

Deleting columns from Dataframe

Use del operator

```
del df['id']
df
```

➔

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Index

First column denotes the index. Pandas Dataframe logically has two Row indexes - One that is shown in Output and can be renamed like we can rename columns; other one which is hidden/cannot be changes and denotes the row position number (0-m)

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
3	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
4	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Visible index-

```
df.index
# Output: RangeIndex(start=0, stop=5, step=1)
```

Accessing elements by visible index-

```
df.loc[[1, 2]]
```

→

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
1	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
2	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990

.loc[] takes the values as in the visible index

Changing the visible index-

```
df.index = ['a', 'b', 'c', 'd', 'e']
df
```

→

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
a	Nissan	Stanza	1991	138.0	4	MANUAL	sedan	2000
b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
c	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
d	GMC	Acadia	2017	194.0	4	AUTOMATIC	4dr SUV	34450
e	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

```
df.loc[['b', 'c']]
```

→

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
c	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990

.loc[] takes the values as in the visible index

Accessing elements by positional index

iloc

```
# Still we can refer to a positional index (what we usually use in lists
df.iloc[[1, 2, 4]]
```

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
b	Hyundai	Sonata	2017	NaN	4	AUTOMATIC	Sedan	27150
c	Lotus	Elise	2010	218.0	4	MANUAL	convertible	54990
e	Nissan	Frontier	2017	261.0	6	MANUAL	Pickup	32340

Resetting the visible index

This keeps the previous index and creates a new column called index.

```
df.reset_index()
```

→

	index	Make	Model	Year	Engine HP
0	a	Nissan	Stanza	1991	138.0
1	b	Hyundai	Sonata	2017	NaN
2	c	Lotus	Elise	2010	218.0
3	d	GMC	Acadia	2017	194.0
4	e	Nissan	Frontier	2017	261.0

If the old index is not needed it can simply be dropped.

```
df = df.reset_index(drop=True)
df
```

→

	Make	Model	Year	Engine HP
0	Nissan	Stanza	1991	138.0
1	Hyundai	Sonata	2017	NaN
2	Lotus	Elise	2010	218.0
3	GMC	Acadia	2017	194.0
4	Nissan	Frontier	2017	261.0

Column-wise operations

'Engine HP' column has one row = "NaN" (not a number). We can apply mathematical operations on columns

```
# NaN denotes a missing number
df['Engine HP']
# Output:
# 0    138.0
# 1      NaN
# 2    218.0
# 3    194.0
# 4    261.0
# Name: Engine HP, dtype: float64
```

,

```
df['Engine HP'] / 100
# Output:
# 0    1.38
# 1     NaN
# 2    2.18
# 3    1.94
# 4    2.61
# Name: Engine HP, dtype: float64
```

Comparison operator-

```
df.Year
# Output:
# 0    1991
# 1    2017
# 2    2010
# 3    2017
# 4    2017
# Name: Year, dtype: int64
```

```
df.Year >= 2015
# Output:
# 0    False
# 1     True
# 2    False
# 3     True
# 4     True
# Name: Year, dtype: bool
```

Filtering rows, columns

Selecting specific rows or columns from a DataFrame based on certain conditions. Multiple techniques in Pandas for filtering

Using Boolean mask

Use boolean indexing. This involves creating a boolean mask that specifies the conditions we want to apply to our data. The mask is a DataFrame or Series of the same shape as the original data, where each element is either True or False depending on whether the corresponding element in the original data satisfies the condition.

Filter out all rows where the year is greater or equal to 2015

```
condition = df.Year >= 2015
```

```
filtered_df = df[condition]
```

OR

```
# or like this
df[df.Year >= 2015]
```

Using .query() method

Filter rows based on a string expression, similar to writing SQL queries.

```
filtered_df = df.query('year >=  2015')
```

Combine several condition, for example, you want to have all Nissans after the year 2015.

```
df[
    (df.Make == 'Nissan') & (df.Year > 2015)
]
```

String operations

Numpy does not have string operations since it processes Numbers. Pandas has this for data cleaning or formatting

Vehicle_style has different casing for same words (sedan, Sedan). It also has spaces which we want to replace with underscore

Vehicle_Style
sedan
Sedan
convertible
4dr SUV
Pickup

```
df['Vehicle_Style'].str.lower()
# Output:
# 0           sedan
# 1           sedan
# 2     convertible
# 3         4dr suv
# 4          pickup
# Name: Vehicle_Style, dtype: object
```

```
df['Vehicle_Style'].str.replace(' ','_')
# Output:
# 0           sedan
# 1           Sedan
# 2     convertible
# 3         4dr_SUV
# 4          Pickup
# Name: Vehicle_Style, dtype: object
```

Both operations can be chained together

```
# Both operations can be chained
df['Vehicle_Style'] = df['Vehicle_Style'].str.lower().str.replace(' ','_')
df
```

→

Vehicle_Style
sedan
sedan
convertible
4dr_suv
pickup

Aggregate (Summarizing) operations

```
df.MSRP
# Output:
# 0     2000
# 1    27150
# 2    54990
# 3    34450
# 4    32340
# Name: MSRP, dtype: int64
```

```
df.MSRP.min()
# Output: 2000

df.MSRP.max()
# Output: 54990

df.MSRP.mean()
# Output: 30186.0
```

Describe() for numerical stats

Provides count, mean, std, min, max etc. Can be applied on single column or on all the numerical columns of the Dataframe

```
df.MSRP.describe()
# Output:
# count         5.000000
# mean      30186.000000
# std       18985.044904
# min        2000.000000
# 25%       27150.000000
# 50%       32340.000000
# 75%       34450.000000
# max       54990.000000
# Name: MSRP, dtype: float64
```

```
df.describe()
```

	Year	Engine HP	Engine Cylinders	MSRP
count	5.000000	4.00000	5.000000	5.000000
mean	2010.400000	202.75000	4.400000	30186.000000
std	11.260551	51.29896	0.894427	18985.044904
min	1991.000000	138.00000	4.000000	2000.000000
25%	2010.000000	180.00000	4.000000	27150.000000
50%	2017.000000	206.00000	4.000000	32340.000000
75%	2017.000000	228.75000	4.000000	34450.000000
max	2017.000000	261.00000	6.000000	54990.000000

```
df.describe().round(2)
```

	Year	Engine HP	Engine Cylinders	MSRP
count	5.00	4.00	5.00	5.00
mean	2010.40	202.75	4.40	30186.00
std	11.26	51.30	0.89	18985.04
min	1991.00	138.00	4.00	2000.00
25%	2010.00	180.00	4.00	27150.00
50%	2017.00	206.00	4.00	32340.00
75%	2017.00	228.75	4.00	34450.00
max	2017.00	261.00	6.00	54990.00

Unique values

Can be applied on single column or on all the columns (numerical, categorical) of the Dataframe

nunique() -> Number of unique values

```
# Returns the number of unique values of column Make
df.Make.nunique()
# Output: 4
```

```
# Returns the number of unique values for all columns of the DataFrame
df.nunique()
# Output:
# Make                4
# Model               5
# Year                3
# Engine HP           4
# Engine Cylinders    2
# Transmission Type   2
# Vehicle_Style       4
# MSRP                5
# dtype: int64
```

unique() → Unique values

```
df.Year.unique()
# Output
array([1991, 2017, 2010])
```

Missing values

isnull()

```
df.isnull()
```

→

	Make	Model	Year	Engine HP	Engine Cylinders	Transmission Type	Vehicle_Style	MSRP
0	False	False	False	False	False	False	False	False
1	False	False	False	True	False	False	False	False
2	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False

The tabular representation is very confusing therefore it's more useful to sum up the number of missing values for each column.

```
df.isnull().sum()
# Output:
# Make                0
# Model               0
# Year                0
# Engine HP           1
# Engine Cylinders    0
# Transmission Type   0
# Vehicle_Style       0
# MSRP                0
# dtype: int64
```

Grouping

Just like SQL has Groupby. If you want to get average price for each Transmission type

```sql
SELECT
    transmission_type,
    AVG(MSRP)
FROM
    cars
GROUP BY
    transmission_type
```

→

```python
df.groupby('Transmission Type').MSRP.mean()
# Output:
# Transmission Type
# AUTOMATIC     30800.000000
# MANUAL        29776.666667
# Name: MSRP, dtype: float64
```

Describe() with groupby()

```python
df.groupby('Transmission Type').MSRP.describe()
```

→

Transmission Type	count	mean	std	min	25%	50%	75%	max
AUTOMATIC	2.0	30800.000000	5161.879503	27150.0	28975.0	30800.0	32625.0	34450.0
MANUAL	3.0	29776.666667	26587.836191	2000.0	17170.0	32340.0	43665.0	54990.0

Get underlying Numpy Array

```python
df.MSRP.values
# Output:
# array([ 2000, 27150, 54990, 34450, 32340])
```

Get/convert to list of dictionaries

```python
df.to_dict(orient='records')
```

→

```python
# Output:
# [{'Make': 'Nissan',
#   'Model': 'Stanza',
#   'Year': 1991,
#   'Engine HP': 138.0,
#   'Engine Cylinders': 4,
#   'Transmission Type': 'MANUAL',
#   'Vehicle_Style': 'sedan',
#   'MSRP': 2000},
# {'Make': 'Hyundai',
#   'Model': 'Sonata',
#   'Year': 2017,
#   'Engine HP': nan,
#   'Engine Cylinders': 4,
#   'Transmission Type': 'AUTOMATIC',
#   'Vehicle_Style': 'sedan',
#   'MSRP': 27150},
# {'Make': 'Lotus',
#   'Model': 'Elise',
#   'Year': 2010,
#   'Engine HP': 218.0,
#   'Engine Cylinders': 4,
#   'Transmission Type': 'MANUAL',
#   'Vehicle_Style': 'convertible',
#   'MSRP': 54990},
# {'Make': 'GMC',
#   'Model': 'Acadia',
#   'Year': 2017,
#   'Engine HP': 194.0,
#   'Engine Cylinders': 4,
#   'Transmission Type': 'AUTOMATIC',
#   'Vehicle_Style': '4dr_suv',
#   'MSRP': 34450},
# {'Make': 'Nissan',
#   'Model': 'Frontier',
#   'Year': 2017,
#   'Engine HP': 261.0,
#   'Engine Cylinders': 6,
#   'Transmission Type': 'MANUAL',
#   'Vehicle_Style': 'pickup',
#   'MSRP': 32340}]
```

2. Project - Car Price Prediction

Implementation of an ML project and which steps have to be considered. It is about the prediction of car prices based on a Kaggle dataset (https://www.kaggle.com/datasets/CooperUnion/cardataset).

Major steps are-

1. Data preparation
2. EDA (Exploratory Data Analysis)
3. Use linear regression for price prediction
   (MSRP – Manufacturer suggested retail price)
4. Understand the internals of linear regression
5. Evaluating the model with RMSE (root mean squared error)
6. Feature Engineering (creating new features)
7. Regularization
8. Using the model

For code and related theory to the code, refer to the Jupyter notebook "2-Regression-CarPricePrediction.ipynb"
Other general theory is in here

Data Preparation

In the data preparation phase, several important steps need to be followed to ensure the dataset is suitable for analysis and modeling. Here are some key considerations:

1. **Data Cleaning**: This involves handling missing values, dealing with outliers, and ensuring consistency in data formats. Missing values can be imputed using various techniques, such as mean/median imputation or using advanced methods like K-nearest neighbors. Outliers may need to be addressed by either removing them or transforming them to fall within a reasonable range.
2. **Data Integration**: If you have multiple datasets related to car prices, you may need to combine them into a single dataset. This can involve matching and merging records based on common identifiers or performing data joins based on shared attributes.
3. **Data Transformation**: Sometimes, the existing variables may not be in a suitable format for analysis. In such cases, feature engineering techniques can be applied to create new variables that may have a better relationship with the target variable, such as transforming categorical variables into numerical ones using one-hot encoding or label encoding.
4. **Feature Scaling**: It is crucial to make sure that the features are on a similar scale to avoid bias in the model. Common techniques for feature scaling include standardization (mean of 0 and standard deviation of 1) or normalization (scaling values between 0 and 1).
5. **Train-Validation Split**: Before building the predictive model, it is essential to split the dataset into training and validating subsets. Typically, the majority of the data is used for training, while a smaller portion is reserved for evaluating the model's performance. As I mentioned in past articles, a train-validate-test split might provide more reliable results.

By following these steps diligently, you can ensure that the data is well-prepared and ready for the subsequent stages of the car price prediction project.

Exploratory data analysis (EDA)

Exploratory data analysis (EDA) is an essential step in the data analysis process. It involves summarizing and visualizing the main characteristics of a dataset to gain insights and identify patterns or trends. By exploring the data, researchers can uncover hidden relationships between variables and make informed decisions.

One common technique in EDA is to calculate summary statistics like mean, median, and standard deviation to understand the distribution of the data. These statistics provide a general overview of the dataset and can help identify potential outliers or unusual patterns.

Visualizations also play a crucial role in EDA. Graphical representations such as histograms, scatter plots, and box plots help visualize the data distribution, identify clusters or groups, and detect any unusual patterns or trends.

Visualizations can be particularly helpful in identifying relationships between variables or finding patterns that may not be immediately apparent.

Another important aspect of EDA is data cleaning. This involves handling missing values, outliers, and inconsistencies in the dataset. By carefully examining the data, researchers can decide how to handle missing values (e.g., imputing or removing them) and identify and address outliers or errors.

EDA is not a one-time process but rather an iterative one. As researchers delve deeper into the data, they may uncover additional questions or areas of interest that require further exploration. Through this iterative process, researchers refine their understanding of the data and uncover valuable insights.

In conclusion, exploratory data analysis is a crucial step in the data analysis process. By summarizing, visualizing, and cleaning the data, researchers can uncover patterns, identify relationships, and make informed decisions. It provides the foundation for more advanced data analysis techniques and helps in the formation of hypotheses for further investigation.

Understanding Linear Regression

Linear regression is a fundamental statistical technique used in the field of machine learning for solving regression problems. In simple terms, regression analysis involves predicting a continuous outcome variable based on one or more input features. That means the output of the model is a number.

In the case of linear regression, the basic idea is to find the best-fitting linear relationship between the input features and the output variable. This relationship is represented by a linear equation of the form:

$$y = w0 + w1x1 + w2x2 + \ldots + wnxn$$

Here, "y" represents the output variable, and x1, x2, …, xn represent the input features. The w0, w1, w2, …, wn are the coefficients that determine the relationship between the input features and the output variable.

The goal of linear regression is to estimate the values of these coefficients in such a way that the sum of squared differences between the observed and predicted values is minimized. This minimization is typically achieved using a method called "ordinary least squares," which calculates the best-fitting line by minimizing the sum of the squared errors between the predicted and actual values.

Simplified understanding

Just to recap what we know from the articles before, there is the function g(X) ~ y with: g as the model (in our example Linear regression), X as the feature matrix and y as the target (in our example price).

Now let's step back and look at only one observation. The corresponding function is $g(x_i) \sim y_i$ with: x_i as one car and y_i as its price. So x_i is one specific row of the feature matrix X. We can think of it as a vector with multiple elements (n different characteristics of this one car).

$$x_i = (x_{i1}, x_{i2}, x_{in}) \rightarrow g(x_{i1}, x_{i2}, x_{in}) \sim y_i$$

Let's look at one example and how this looks in code.

```
df_train.iloc[10]

# Output:
# make                      chevrolet
# model                         sonic
# year                           2017
# engine_fuel_type    regular_unleaded
# engine_hp                     138.0
# engine_cylinders                4.0
# transmission_type         automatic
# driven_wheels       front_wheel_drive
# number_of_doors                 4.0
# market_category                 NaN
# vehicle_size                compact
# vehicle_style                 sedan
# highway_mpg                      34
# city_mpg                         24
# popularity                     1385
# Name: 10, dtype: object
```

We take as an example the characteristic enging_hp, city_mpg, and popularity.
xi = [138, 24, 1385]
That's almost everything we need to implement $g(x_i) \sim y_i$:

- x_i = (138, 24, 1385)
- with i = 10
- need to implement the function $g(x_{i2}, x_{i2}, \dots, x_{in}) \sim y_i$

```
# in code this would look like --> this is what we want to implement

def g(xi):
    # do something and return the predicted price
    return 10000

g(xi)
# Output: 10000
```

However, this function g is still not very useful, because it always returns a fixed price. We need to implement the function

$$g(x_i) = w_0 + w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3}$$

with w_0 as bias term and w_1, w_2, and w_3 as weights. This formula can be written as

$$g(x_i) = w_0 + \sum_{j=1}^{3} w_j x_{ij}$$

Implementation of Linear Regression Function

In general and because of array implementation in Python (indices of arrays start with 0 instead of 1), the formula for linear regression looks like this:

$$g(x_i) = w_0 + \sum_{j=0}^{n-1} w_j x_{ij}$$

The following snippet shows the implementation of the g-function (renamed as linear_regression).

```
def linear_regression(xi):
    n = len(xi)
    pred = w0

    for j in range(n):
        pred = pred + w[j] * xi[j]

    return pred

# sample values for w0 and w and the given xi
xi = [138, 24, 1385]
w0 = 0
w = [1, 1, 1]

linear_regression(xi)
# Output: 1547

# try some other values
w0 = 7.17
w = [0.01, 0.04, 0.002]
linear_regression(xi)
# Output: 12.280000000000001
```

What does this actually mean?

We've just implemented the formula as mentioned before with given values:
7.17 + 138*0.01 + 24*0.04 + 1385*0.002 = 12.28

- w_0 = 7.17 bias term = the prediction of a car, if we don't know anything about this
- engine_hp: 138 * 0.01 that means in this case per 100 hp the price will increase by $1
- city_mpg: 24 * 0.04 that means analog to hp, the more gallons the higher the price will be
- popularity: 1385 * 0.002 analog, but it doesn't seem that it's affecting the price too much, so for every extra mention on twitter the car becomes just a little bit more expensive

There is still one important step to do. Because we logarithmized (log(y+1)) the price at the beginning, we now have to undo that. This gives us the predicted price in $. Our car has a price of $215,344.72.

```
# Get the real prediction for the price in $
# We do "-1" here to undo the "+1" inside the log
np.exp(12.280000000000001) - 1
# Output: 215344.7166272456

# Shortcut to not do -1 manually
np.expm1(12.280000000000001)
# Output: 215344.7166272456

# Just for checking only
np.log1p(215344.7166272456)
# Output: 12.280000000000001
```

Simplified form to Vector Form

Continuing from simplified form, we generalize it to Vector form
That means coming back from only one observation x_i (of one car) to the whole feature matrix X.

$$g(x_i) = w_0 + \sum_{j=0}^{n-1} w_j \, x_{ij}$$

The last part of this formula can be rewritten as vector-vector multiplication Or dot product of vectors w & xi

$g(x_i) = w0 + x_i^T w$

Why transpose a vector?

This transpose notation is used only for mathematical notation because in mathematics, a vector is a 1-column array of dimensions n x 1. So, they are written as follows -

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} \quad \begin{bmatrix}4\\5\\6\\7\end{bmatrix}$$

Now, we can multiply these two vectors since their 'n' is same but before we can do that the no. of columns of the 1st vector should match with no of rows of 2nd vector. So, we write 1st vector-transpose and then multiply it

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} . \begin{bmatrix}4\\5\\6\\7\end{bmatrix} = [4+10+18+28] = [60]$$

Note: In code, we do not do any explicit transpose for vector-vector, vector-matrix, matrix-matrix multiplication

Continuing original discussion, we can rewrite the linear_regression() function as follows-

```
def dot(xi, w):
    n = len(xi)

    res = 0.0
    for j in range(n):
        res = res + xi[j] * w[j]

    return res
```

,

```
def linear_regression(xi):
    return w0 + dot(xi, w)
```

To make the last equation more simple, we can imagine there is one more feature x_{i0}, that is always equal to 1.

$$g(x_i) = w_0 + x_i^T w \rightarrow g(x_i) = w_0 x_{i0} + x_i^T w \text{ where } x_{i0} = 1$$

To include the w0 also as part of the w vector, we can consider that there is an additional 1 at the beginning of Feature X matrix. And all the vectors now become (n+1) dimension

- $w = [w_0, w_1, w_2, \ldots w_n]$
- $x_i = [x_{i0}, x_{i1}, x_{i2}, \ldots, x_{in}] = [1, x_{i1}, x_{i2}, \ldots, x_{in}]$
- $w^T x_i = x_i^T w = w_0 + \ldots$

That means we can use the dot product for the entire regression and we can update the linear_regression function as below-

```
xi = [138, 24, 1385]
w0 = 7.17
w = [0.01, 0.04, 0.002]

# adding w0 to the vector w
w_new = [w0] + w
w_new
# Output: [7.17, 0.01, 0.04, 0.002]

xi
# Output: [138, 24, 1385]
```

```
def linear_regression(xi):
    xi = [1] + xi
    return dot(xi, w_new)

linear_regression(xi)
# Output: 12.280000000000001
```

Vector form to Matrix Form

Now, we consider multiple vectors for x i.e. Matrix X is a m x (n+1) dimensional matrix (with m rows and n+1 columns). Each row in matrix X is a vector. For each row of X, we multiply this row with the vector w. i.e. we are doing vector-vector multiplication for each row and the result will be a (m x 1) vector which contains our predictions, and we call it y_{pred}.

$$X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & x_{21} & \dots & x_{2n} \\ 1 & x_{31} & \dots & x_{3n} \\ \dots & \dots & \dots & \dots \\ 1 & x_{m1} & \dots & x_{mn} \end{bmatrix} \qquad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \dots \\ w_n \end{bmatrix}$$

$$Xw = \begin{bmatrix} x_1^T w \\ x_2^T w \\ x_3^T w \\ \dots \\ x_m^T w \end{bmatrix} = y_{pred}$$

So, we can implement Matrix-vector multiplication of X.w in terms of vector-vector multiplication.
Refer figure below for understanding - We define some vectors x1, x2, x10 as lists. Make X as numpy array(2-d array or matrix) from x1, x2, x10. And then use the numpy .dot() method to multiply X with w_new. This gives the predictions (log values) in y vector. Log values converted to actual values using np.expm1() function

```
w0 = 7.17
w = [0.01, 0.04, 0.002]
w_new = [w0] + w

x1  = [1, 148, 24, 1385]
x2  = [1, 132, 25, 2031]
x10 = [1, 453, 11, 86]

# X becomes a list of lists
X = [x1, x2, x10]
X
# Output: [[1, 148, 24, 1385], [1, 132, 25, 2031], [1, 453, 11, 86]]

# This turns the list of lists into a matrix
X = np.array(X)
X
# Output:
# array([[   1,  148,   24, 1385],
#        [   1,  132,   25, 2031],
#        [   1,  453,   11,   86]])

# Now we have predictions, so for each car we have a price for this car
y = X.dot(w_new)

# shortcut to not do -1 manually to get the real $ price
np.expm1(y)
# Output: array([237992.82334859, 768348.51018973, 222347.22211011])
```

Based on the above understanding, we update linear_regression function as follows-
So, it basically takes a matrix X as parameter and uses numpy.dot() function

```
def linear_regression(X):
    return X.dot(w_new)

y = linear_regression(X)
np.expm1(y)
# Output: array([237992.82334859, 768348.51018973, 222347.22211011])
```

For the understanding discussions till now, the w_new (weight vector) had some arbitrary values. But, the actual values of w_new will come from training the model

Training the linear regression model

This means given the Feature Matrix X and Target Vector y, we have to calculate the Weights vector w.

Ideally, we want $X.w = y$ i.e. exact values. But, in reality,

$$X.w \approx y$$

So, we need to solve this equation for w

Now, assume the matrix X is invertible [An **invertible** matrix is a **square** matrix whose **inverse exists** and when multiplied by its inverse, the result is the identity matrix]
ie. $X^{-1}.X = X.X^{-1} = I$
Recall that inverse(X) = adjoint (X) / determinant (X). Determinant exists (i.e. we can calculate determinant) only for square matrices and so inverse exists only for square matrices

Multiply both sides of model equation with X-inverse (X^{-1})

$$X^{-1}.X.w = X^{-1}.y$$

Replace Xinv.X with Identity matrix

$$I.w = X^{-1}.y$$

And Identity matrix is equivalent to number 1 in simple algebra, so I.w = w

$$w = X^{-1}.y$$

Now, if X was always a square matrix, then we can simply stop here and get w. But, in real world, the feature matrix is a rectangular matrix with no of rows >> no of columns i.e. no of observation >> no of features. So, for rectangular matrices, determinant does not exist → inverse does not exist → so we need to make X a square matrix

Now, for matrix A with order (m x n), to make it square, we can multiply it with its transpose A-transpose (n x m)
So, At.A => n x n square matrix and A.At => m x m square matrix

Lets multiply original equation X.w = y with Xt

$$X^T X.w \approx X^T.y$$

The matrix X^TX is called **GRAM MATRIX** and the inverse exists. Order of the Gram matrix will be (n+1 X n+1)

Multiply both sides with $(X^TX)^{-1}$ to reduce it to Identity matrix and then 1

$$(X^T X)^{-1}.(X^T X).w \approx (X^T X)^{-1}.X^T.y$$

$$I.w \approx (X^T X)^{-1}.X^T.y$$

$$w \approx (X^T X)^{-1}.X^T.y$$

This “w” will have the closest values [Refer to another book “Elements of Statistical Learning” for proof of this if interested]. This is called the NORMAL EQUATION of LINEAR REGRESSION

So, we need to implement this Normal Equation in function train_linear_regression(), that takes the feature matrix X and the target variable y and returns the vector w.

Consider feature matrix X with data of 10 cars and 3 features

```
X =[
    [148, 24, 1385],
    [132, 25, 2031],
    [453, 11, 86],
    [158, 24, 185],
    [172, 25, 201],
    [413, 11, 83],
    [38, 54, 185],
    [142, 25, 431],
    [453, 31, 86],
]

X = np.array(X)
X
# Output:
# array([[ 148,   24, 1385],
#        [ 132,   25, 2031],
#        [ 453,   11,   86],
#        [ 158,   24,  185],
#        [ 172,   25,  201],
#        [ 413,   11,   83],
#        [  38,   54,  185],
#        [ 142,   25,  431],
#        [ 453,   31,   86]])
```

To include the bias term w0, we need to add a new column with ones to the feature matrix X for the multiplication with w_0. We remember that we can use np.ones() to get a vector with ones at each position.

```
ones = np.ones(9)
ones
# Output: array([1., 1., 1., 1., 1., 1., 1., 1., 1.])

# X.shape[0] looks at the number of rows and creates the vector of ones
ones = np.ones(X.shape[0])
ones
# Output: array([1., 1., 1.])
```

Now we need to stack this vector of ones with our feature matrix X. For this we can use the NumPy function np.column_stack() as shown in the next snippet.

```
np.column_stack([ones, ones])
# Output:
# array([[1., 1.],
#             [1., 1.],
#             [1., 1.],
#             [1., 1.],
#             [1., 1.],
#             [1., 1.],
#             [1., 1.],
#             [1., 1.],
#             [1., 1.]])


X = np.column_stack([ones, X])
y = [10000, 20000, 15000, 25000, 10000, 20000, 15000, 25000, 12000]

# GRAM MATRIX
XTX = X.T.dot(X)

# Inverse GRAM MATRIX
XTX_inv = np.linalg.inv(XTX)
```

In the following code snippet we test whether the multiplication of XTX with XTX_inv actually produces the Identity matrix I.

```
# Without round(1) it's not exactly identity matrix but the other values
# are very close to 0 --> we can treat them as 0 and take it as identity ma
XTX.dot(XTX_inv)
# Output:
#array([[ 1.00000000e+00,  2.60208521e-18,  4.85722573e-17,
         1.08420217e-18],
#       [ 4.54747351e-13,  1.00000000e+00,  1.50990331e-14,
         2.22044605e-16],
#       [ 5.68434189e-14,  1.11022302e-16,  1.00000000e+00,
         3.46944695e-17],
#       [ 9.09494702e-13,  0.00000000e+00, -2.13162821e-14,
         1.00000000e+00]])

# This gives us the I matrix
XTX.dot(XTX_inv).round(1)
# Output:
# array([[ 1.,  0.,  0.,  0.],
#             [ 0.,  1.,  0.,  0.],
#             [ 0.,  0.,  1.,  0.],
#             [ 0.,  0., -0.,  1.]])
```

Now we can implement the formula to get the full w vector.

```
# w_full contains all the weights
w_full = XTX_inv.dot(X.T).dot(y)
w_full
# Output: array([ 3.00092529e+04, -2.27839691e+01, -2.57690874e+02, -2.3032
```

From that vector w_full we can extract w0 and all the other weights.

```
w0 = w_full[0]
w = w_full[1:]
w0, w
# Output: (30009.25292276666, array([ -22.78396914, -257.69087426,   -2.303
```

Now we can implement the function train_linear_regression, that takes the feature matrix X and the target variable y and returns w_0 and the vector w.

```python
def train_linear_regression(X, y):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    XTX_inv = np.linalg.inv(XTX)
    w_full = XTX_inv.dot(X.T).dot(y)

    return w_full[0], w_full[1:]
```

Let's test this newly implemented function with some simple examples:

```python
X =[
    [148, 24, 1385],
    [132, 25, 2031],
    [453, 11, 86],
    [158, 24, 185],
    [172, 25, 201],
    [413, 11, 83],
    [38, 54, 185],
    [142, 25, 431],
    [453, 31, 86],
]

X = np.array(X)
y = [10000, 20000, 15000, 25000, 10000, 20000, 15000, 25000, 12000]

train_linear_regression(X, y)
# Output: (30009.25292276666, array([ -22.78396914, -257.69087426,   -2.303
```

Refer the Jupyter Notebook "Car Price Prediction.ipynb" Section 2.7 for above implementation

Car Price Baseline Model

Here, we build a baseline model for price prediction of a car. Here we'll use the implemented code from the last article to build the model. First we start with a simple model while we're using only **numerical columns**.

Value Extraction

The next code snippet shows how to extract all numerical columns. By the way you can also use it to extract categorical columns.

```
df_train.dtypes

# Output:
# make                  object
# model                 object
# year                   int64
# engine_fuel_type      object
# engine_hp            float64
# engine_cylinders     float64
# transmission_type     object
# driven_wheels         object
# number_of_doors      float64
# market_category       object
# vehicle_size          object
# vehicle_style         object
# highway_mpg            int64
# city_mpg               int64
# popularity             int64
# dtype: object

df_train.columns
# Output:
# Index(['make', 'model', 'year', 'engine_fuel_type', 'engine_hp',
#        'engine_cylinders', 'transmission_type', 'driven_wheels',
#        'number_of_doors', 'market_category', 'vehicle_size', 'vehicle_styl
#        'highway_mpg', 'city_mpg', 'popularity'],
#       dtype='object')
```

We choose the columns **engine_hp, engine_cylinders, highway_mpg, city_mpg,** and **popularity** for our base model.

```
base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg', 'popular

df_train[base].head()
```

	engine_hp	engine_cylinders	highway_mpg	city_mpg	popularity
0	310.0	8.0	18	13	1851
1	170.0	4.0	32	24	640
2	165.0	6.0	15	13	549
3	150.0	4.0	39	28	873
4	510.0	8.0	19	13	258

Value extraction

We need to extract the values to use them in training.

```
X_train = df_train[base].values
X_train
# Output:
#array([[ 310.,    8.,   18.,   13., 1851.],
#            [ 170.,    4.,   32.,   24.,  640.],
#            [ 165.,    6.,   15.,   13.,  549.],
#            ...,
#            [ 342.,    8.,   24.,   17.,  454.],
#            [ 170.,    4.,   28.,   23., 2009.],
#            [ 160.,    6.,   19.,   14.,  586.]])
```

Handling missing values

Missing values are generally not good for our model. Therefore, you should always check whether such values are present.

```
df_train[base].isnull().sum()
# Output:
# engine_hp           42
# engine_cylinders    17
# highway_mpg          0
# city_mpg             0
# popularity           0
# dtype: int64
```

As you can see there are two columns that have missing values. The easiest thing we can do is fill them with zeros. But notice filling it with 0 makes the model ignore this feature, because:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2$$

if $x_{i1} = 0$ then the previous equation simplifies to

$$g(x_i) = w_0 + 0 + x_{i2}w_2$$

But 0 is not always the best way to deal with missing values, because that means there is an observation of a car with 0 cylinders or 0 horse powers. And a car without cylinders or 0 horse powers does not make much sense at this point. However, for the current example, we will set to 0.

```
df_train[base].fillna(0).isnull().sum()
# Output:
# engine_hp           0
# engine_cylinders    0
# highway_mpg         0
# city_mpg            0
# popularity          0
# dtype: int64
```

As you can see in the last snippet, the missing values have disappeared. However, now we need to apply this change in the DataFrame.

```
X_train = df_train[base].fillna(0).values
X_train
# Output:
# array([[ 310.,    8.,   18.,   13., 1851.],
#             [ 170.,    4.,   32.,   24.,  640.],
#             [ 165.,    6.,   15.,   13.,  549.],
#             ...,
#             [ 342.,    8.,   24.,   17.,  454.],
#             [ 170.,    4.,   28.,   23., 2009.],
#             [ 160.,    6.,   19.,   14.,  586.]])

y_train
# Output:
# array([10.40262514, 10.06032035,  7.60140233, ..., 10.92181127,
#        9.91100919,  8.10892416])
```

Now we can train our model using the train_linear_regression function that we've implemented in the last article. The function return the value for w_0 and and array for vector w.

```
w0, w = train_linear_regression(X_train, y_train)
w0, w
# Output:
# (7.471835414587793,
#  array([ 9.30959186e-03, -1.19533938e-01,  4.68925224e-02, -1.13441722e-02,
#         -1.01631104e-05]))
```

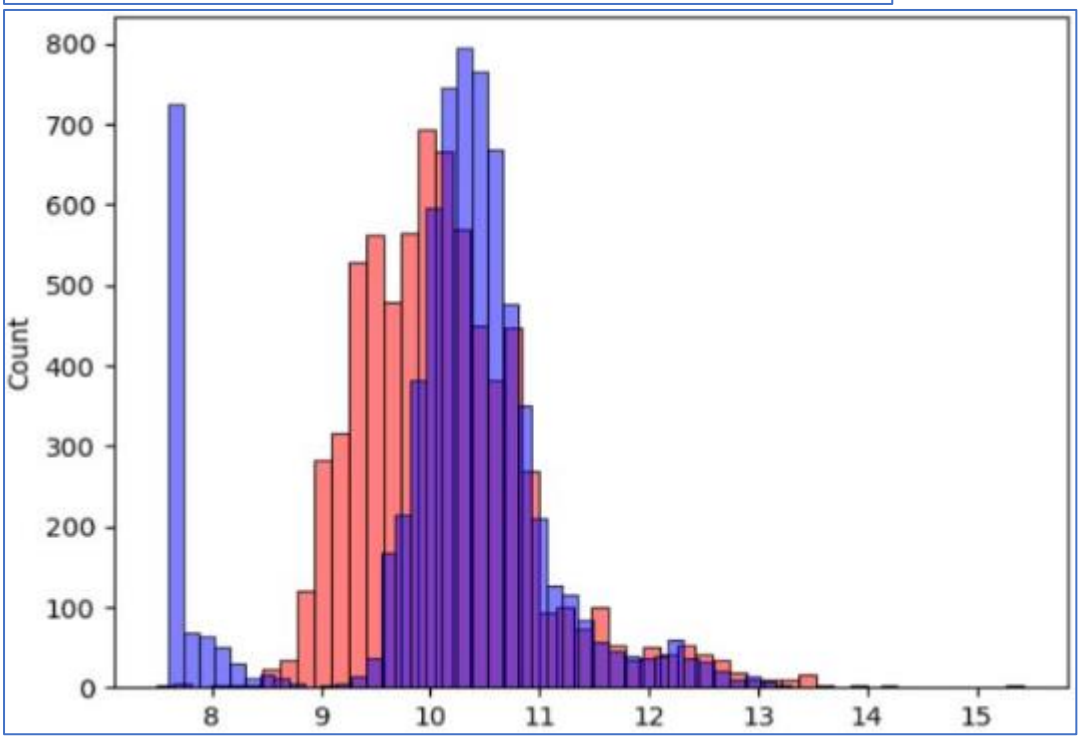
We can use this two variables to apply the model to our training dataset to see how well the model has learned the training data. We do not want the model to simply memorize the data, but to recognize the correlations. Later, we'll also apply the model to unseen data so that we can eliminate memorization.

```
y_pred = w0 + X_train.dot(w)
y_pred
# Output:
# array([10.07931663,  9.79812647,  8.84104849, ..., 10.62739988,
        9.60798726,  8.97035041])
```

Plotting model performance

Three snippets before, you can see the actual values for y. The last snippet shows the predicted values for y. You could now go through these two lists manually and compare them. A better and also visual possibility is provided by Seaborn. The next snippet shows how to output these two lists accordingly.

```
# alpha changes the transparency of the bars
# bins specifies the number of bars
sns.histplot(y_pred, color='red', alpha=0.5, bins=50)
sns.histplot(y_train, color='blue', alpha=0.5, bins=50)
```



You see from this plot that the model is not ideal but it's better to have an objective way of saying that the model is good or not good. When we start improving the model, we also want to ensure that we really improving it. The next article is about an objective way to evaluate the performance of a regression model.

Evaluating the model with RMSE (root mean squared error)

RMSE is to quantify how close/far the predicted values are to the expected values. It measures the error associated with the model being evaluated. This numerical figure can then be used to compare models, enabling us to choose the one that gives the best predictions

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(g(x_i) - y_i)^2}$$

- g(xi) is the prediction
- yi is the actual value
- m is the number of observations in the dataset (i.e. cars)

g(xi) - yi is the error. We square it to get square error. Then, we take the mean of the square errors and then we do the square root

y_pred	10	9	11	...	10
y_train	9	9	10.5	...	11.5
y_pred – y_train	1	0	0.5	...	-1.5

square the difference: $(g(xi) - yi)^2$	1	0	0.25	...	2.25	SQUARED ERROR

average	(1 + 0 + 0.25 + 2.25) / 4 = 0.875	MEAN SQUARED ERROR
root	sqrt(0.875) = 0.93	ROOT MEAN SQUARED ERROR

In code, the function rmse() is-

```
def rmse(y, y_pred):
    se = (y - y_pred) ** 2
    mse = se.mean()
    return np.sqrt(mse)
```

```
rmse(y_train, y_pred)
# Output: 0.7464137917148924
```

In the example above, we found RMSE by comparing the Predicted values with Training output. That was only for understanding. In practice, we would compute RMSE by comparing the Predicted values with Validation output (y-val)

Evaluating the model performance on the training data does not really give a good indication of the real model performance. Since we don't know how well the model can apply the learned knowledge to unseen data. So what we want to do now after training the model g on our training dataset, we want to apply it on the validation dataset to see how it performs on unseen data. We use RMSE for validating the performance.

```
base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg', 'popular

X_train = df_train[base].fillna(0).values
```

If you see the above snippet, we need to repeat this for Xval, Xtest. So, instead create a common function, prepare_x() to select the features and clean the data

```
def prepare_X(df):
    df_num = df[base]   # Take only the features(columns) of our interest
    df_num = df_num.fillna(0)   # Handle missing values
    X = df_num.values   # Extract the matrix as a Numpy 2D Array
    return X
```

Now we can use this function when we prepare data for the training and for the validation as well. In the training part we only use training dataset to train the model. In the validation part we prepare the validation dataset the

same way like before and apply the model. Lastly we compute the rmse.

```
## Prepare the input Training Feature matrix
X_train = prepare_X(df_train)

## Make the model learn on the training data i.e. compute the weights w0, w vector
w0, w = train_linear_regression(X_train, y_train)

## Prepare the input Validation Feature matrix
X_val = prepare_X(df_val)

## Predict the output using Xval Validation matrix
y_pred = w0 + X_val.dot(w)

## Compute the RMSE of Validation Predicted and Validation Expected values
rmse(y_val, y_pred)
```

✓ 0.0s

```
0.7616530991301607
```

When we compare the RMSE from training with the value from validation (0.746 vs. 0.7616) we see that the model performs similarly well on the seen and unseen data. That is what we have hoped for.

Feature Engineering (Including more features to improve model performance)

Till now, we used 5 features as the base and that was our baseline model. To improve the performance of this model, we should add more features to the model. We know that for cars, the age of the car is key factor to determine its price. The older the car, the lower the price. So, lets add the age to the feature list. We can calculate the age as (Date of Dataset creation) - (Year of Mfg). Date of Dataset creation is 2017.

```
base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg', 'popula
```

We add new features in the prepare_X() function

```
## Lets add age of car as additional feature to the Input Matrix X

def prepare_X(df):
    df = df.copy()  # Create copy of dataframe so that original dataframe is not altered

    df['age'] = 2017 - df['year']   # Add age column to the copy
    features = base + ['age']   # Add age with base to the features list

    df_num = df[features]   # Extract the feature columns
    df_num = df_num.fillna(0)   # Handle missing values in the extracted feature columns
    X = df_num.values   # Extract the feature values

    return X
```

```
## Set the X_train input matrix
X_train = prepare_X(df_train)
X_train
```

✓ 0.0s

age

```
array([[1.480e+02, 4.000e+00, 3.300e+01, 2.400e+01, 1.385e+03, 9.000e+00],
       [1.320e+02, 4.000e+00, 3.200e+01, 2.500e+01, 2.031e+03, 5.000e+00],
       [1.480e+02, 4.000e+00, 3.700e+01, 2.800e+01, 6.400e+02, 1.000e+00],
       ...,
       [2.850e+02, 6.000e+00, 2.200e+01, 1.700e+01, 5.490e+02, 2.000e+00],
       [5.630e+02, 1.200e+01, 2.100e+01, 1.300e+01, 8.600e+01, 3.000e+00],
       [2.000e+02, 4.000e+00, 3.100e+01, 2.200e+01, 8.730e+02, 0.000e+00]])
```

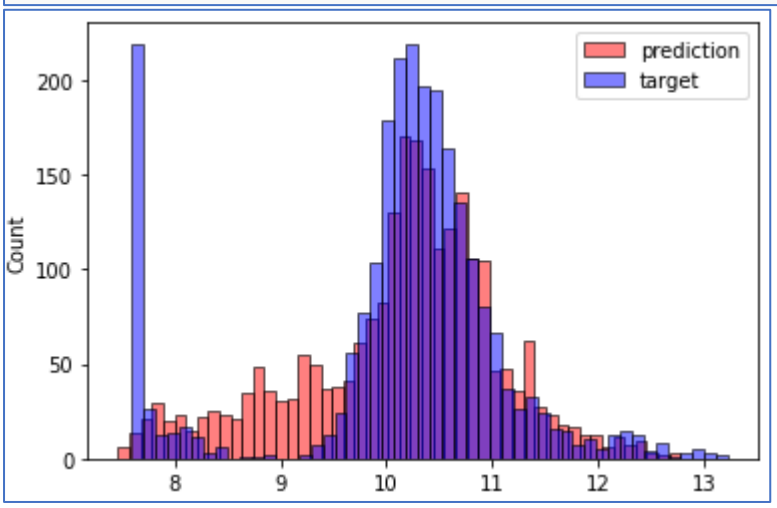
```
## Train the model
w0, w = train_linear_regression(X_train, y_train)

## Evaluate the model against Validation set
X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)

## Calculate RMSE of Prediction vs Validation expected set
rmse(y_val, y_pred)
```

✓ 0.0s

```
0.5172055461058338
```

See that the RMSE dropped from 0.7 to 0.5 → This means our model improved by adding the new feature, age

```
## Visualizing the improvement in model prediction
sns.histplot(y_pred, label='prediction', color='red', alpha=0.5, bins=50)
sns.histplot(y_val, label='target', color='blue',  alpha=0.5, bins=50)
plt.legend()
## The prediction is nicely following the bell curve of the target.
## But, we still have areas of improvement where the prediction is far away from expected
```



Visually, a clear improvement can be seen. Many car prices are predicted much better. But there is still space for improvement.

Handling Categorical Features

Categorical variables are variables that are categories (typically strings)

```
# Inspect the datatypes of columns to identify categorical variables
df_train.dtypes
```
✓ 0.0s

```
make                object
model               object
year                 int64
engine_fuel_type    object
engine_hp          float64
engine_cylinders   float64
transmission_type   object
driven_wheels       object
number_of_doors    float64
market_category     object
vehicle_size        object
vehicle_style       object
highway_mpg          int64
city_mpg             int64
popularity           int64
dtype: object
```

Here: make, model, engine_fuel_type, transmission_type, driven_wheels, market_category, vehicle_size, vehicle_style But, there is one value that looks like numerical variable, but it isn't.
number_of_doors is not really a numerical number but is a category like 2-door car, 3-door car, 4-door car etc

Now, to handle categorical column, we create additional columns = unique values it has in the dataset. So, in this case, there are 3 unique values and so we have 3 new columns and we put 1 in the column matching the number-of-door and 0 in other for each datarow → This is called ONE HOT ENCODING

Num of doors	num_doors_2	num_doors_3	num_doors_4
2	1	0	0
3	0	1	0
4	0	0	1
2	1	0	0

We add these additional columns to the Feature matrix X

To do ONE HOT ENCODING, we can use comparison operator (==) which will generate Boolean True/False.

```
df_train.number_of_doors == 2
```
✓ 0.0s

```
0        True
1       False
2       False
3       False
4       False
        ...
7145     True
7146     True
7147    False
7148    False
7149    False
Name: number_of_doors, Length: 7150
```

```
## EXAMPLE-Add new column for each number of door
df_train['num_doors_2'] = (df_train.number_of_doors == 2).astype('int')
df_train['num_doors_3'] = (df_train.number_of_doors == 3).astype('int')
df_train['num_doors_4'] = (df_train.number_of_doors == 4).astype('int')
```

We do this for each new column and convert the Boolean to int (i.e. 0 or 1).

We can replace individual encoding statements with string replacement for loop

```
'num_doors_%s' % 4
# Output: 'num_doors_4'
```

```
## Example With that replacement we can write a loop and add the new columns with encoding
for v in [2, 3, 4]:
    df_train['num_doors_%s' % v] = (df_train.number_of_doors == v).astype('int')
```

Add the encoding logic in prepare_X()

```
# We apply the encoding logic for "number of doors" in prepare_X()
def prepare_X(df):
    df = df.copy()

    df['age'] = 2017 - df['year']
    features = base + ['age']

    ## One-hot encoding For number of doors
    for v in [2, 3, 4]:
        df['num_doors_%d' % v] = (df.number_of_doors == v).astype(int)
        features.append('num_doors_%d' % v)

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values

    return X
```

```
prepare_X(df_train)
```
✓ 0.0s

```
array([[148.,   4.,  33., ...,   1.,   0.,   0.],
       [132.,   4.,  32., ...,   0.,   0.,   1.],
       [148.,   4.,  37., ...,   0.,   0.,   1.],
       ...,
       [285.,   6.,  22., ...,   0.,   0.,   1.],
       [563.,  12.,  21., ...,   0.,   0.,   1.],
       [200.,   4.,  31., ...,   0.,   0.,   1.]])
```

Train the model with the additional categorical feature, number of doors and check the RMSE

```
## Train the model with categorical variable, number of doors
X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)
rmse(y_val, y_pred)
```
✓ 0.0s

```
0.515799564150262
```

RMSE has insignificantly improved from 0.5172 to 0.5157 meaning that this column is not that influencing the prediction

So, lets take another categorical variable, make. It has 48 values. So, we take the top 5 makes i.e. add 5 new columns for those makes and one-hot-encode them. We add this logic to prepare_X()

```
# Top 5 makes into list
makes = list(df_train['make'].value_counts().head().index)
makes
```

✓ 0.0s

```
['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']
```

```
# We apply the encoding logic for "make" in prepare_X()
def prepare_X(df):
    df = df.copy()

    df['age'] = 2017 - df['year']
    features = base + ['age']

    ## One-hot encoding For number of doors
    for v in [2, 3, 4]:
        df['num_doors_%d' % v] = (df.number_of_doors == v).astype(int)
        features.append('num_doors_%d' % v)

    for v in makes:
        df['make_%s' % v] = (df.make == v).astype(int)
        features.append('make_%s' % v)

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values

    return X
```

And, again train the model on new feature input with ‘make’ and check its RMSE

```
## Train model with 'make' features also and check its RMSE
X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)
rmse(y_val, y_pred)
```

✓ 0.0s

```
0.50760388495572
```

RMSE improved to 0.507 from 0.5157. So, ‘make’ improves the model as we expected

Now, lets add general logic to add top 5 of all the remaining categorical variables and our expectation is that the model should improve significantly

```
## Add all the categorical variables
categorical_columns = [
    'make', 'model', 'engine_fuel_type', 'driven_wheels', 'market_category',
    'vehicle_size', 'vehicle_style']

## Capture each categorical column as key and its top 5 values as Values in dictionary
categorical = {}
for c in categorical_columns:
    categorical[c] = list(df_train[c].value_counts().head().index)
```

```
categorical
```

✓ 0.0s

```
{'make': ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge'],
 'model': ['silverado_1500', 'tundra', 'f-150', 'sierra_1500', 'tacoma'],
 'engine_fuel_type': ['regular_unleaded',
  'premium_unleaded_(required)',
  'premium_unleaded_(recommended)',
  'flex-fuel_(unleaded/e85)',
  'diesel'],
 'driven_wheels': ['front_wheel_drive',
  'rear_wheel_drive',
  'all_wheel_drive',
  'four_wheel_drive'],
 'market_category': ['crossover',
  'flex_fuel',
  'luxury',
  'hatchback',
  'luxury,performance'],
 'vehicle_size': ['compact', 'midsize', 'large'],
 'vehicle_style': ['sedan',
  '4dr_suv',
  'coupe',
  'convertible',
  '4dr_hatchback']}
```

Add the logic of One-hot encoding for all categorical columns in prepare_X()

```python
## Add the logic of One-hot encoding for all categorical columns here
def prepare_X(df):
    df = df.copy()

    df['age'] = 2017 - df['year']
    features = base + ['age']

    for v in [2, 3, 4]:
        df['num_doors_%d' % v] = (df.number_of_doors == v).astype(int)
        features.append('num_doors_%d' % v)

    for name, values in categorical.items():
        for value in values:
            df['%s_%s' % (name, value)] = (df[name] == value).astype(int)
            features.append('%s_%s' % (name, value))

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values

    return X
```

And, train the model and check the RMSE

```python
X_train = prepare_X(df_train)
w0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)
rmse(y_val, y_pred)
```

✓ 0.1s

```
416.8831235468104
```

```
w0, w
```

✓ 0.0s

```
(-676260605536135.5,
 array([-2.31618135e+00, -5.76071434e+01, -2.42477255e+01, -2.69964739e+01,
        -9.87744658e-03, -2.25493039e+01, -2.03920155e+04, -2.07483525e+04,
        -2.05310329e+04, -1.79150239e+01, -5.01139850e+01,  6.61376804e+00,
        -7.27371782e+00, -4.57447468e+01, -1.58189923e+02,  1.47084995e+01,
```

RMSE instead of improving has drastically degraded. And values of w0, w vector are also either very low or very high
So, what went wrong ???

CONCEPT - Added more features and model performance degraded

Remember the normal equation for linear regression co-efficients

$$w \approx (X^T X)^{-1}.X^T.y$$

X^TX is the GRAM MATRIX and if the values in any two columns are exactly same or exactly factor-multiple, then by matrix rules, we need to ignore one of the columns (The column are said to be linear combination of each other). After ignoring, the GRAM Matrix is no longer a square matrix and hence its inverse does not exist. Due to this, the above equation fails and w vector gets random very high/low values

Lets understand this with simple example by having duplicate values in 2 columns-

```
X = [
    [4, 4, 4],
    [3, 5, 5],
    [5, 1, 1],
    [5, 4, 4],
    [7, 5, 5],
    [4, 5, 5],
]
```

```
X = np.array(X)
X
# Output:
# array([[4, 4, 4],
#        [3, 5, 5],
#        [5, 1, 1],
#        [5, 4, 4],
#        [7, 5, 5],
#        [4, 5, 5]])
```

```
XTX = X.T.dot(X)
XTX
# Output:
# array([[140, 111, 111],
#        [111, 108, 108],
#        [111, 108, 108]])
```

We see that columns 2,3 in XTX are also duplicated and then when we try to take inverse of XTX, it gives error "Singular matrix" which means it is not invertible

```
np.linalg.inv(XTX)
# Output: LinAlgError: Singular matrix
```

In our model, we did not get this error so it is not the case of exact duplicate columns or that inverse of GRAM Matrix does not exist. But, we got weird results because the data is not clean (it has some noise and leads to near-duplicate values)

So, lets take the same example but we add some noise to one value in duplicate column in X

```
X = [
    [4, 4, 4],
    [3, 5, 5],
    [5, 1, 1],
    [5, 4, 4],
    [7, 5, 5],
    [4, 5, 5.0000001],
]

X = np.array(X)
y = [1, 2, 3, 1, 2, 3]
```

```
XTX = X.T.dot(X)
XTX
# Output:
# array([[140.       , 111.       , 111.0000004],
#        [111.       , 108.       , 108.0000005],
#        [111.0000004, 108.0000005, 108.000001 ]])

XTX_inv = np.linalg.inv(XTX)
XTX_inv
# Output:
# array([[ 3.93617174e-02, -1.76703046e+05,  1.76703004e+05],
#        [-1.76703046e+05,  4.02107113e+13, -4.02107110e+13],
#        [ 1.76703004e+05, -4.02107110e+13,  4.02107106e+13]])
```

We do the XTX and get the XTX_inverse. We also get the w vector

```
w = XTX_inv.dot(X.T).dot(y)
w
# Output: array([ 2.85838502e-01, -5.04106388e+06,  5.04106425e+06])
```

So, we see that the values are way too low or way too high just due to an insignificant noise

REGULARIZATION (RIDGE REGRESSION)

To overcome the problem of near-duplicate values, we add a small number called alpha to the diagonal elements of XTX GRAM MATRIX. This process is called REGULARIZATION

Example - We add 0.0001 to diagonal elements

```
# Adding a small number to the diagonal
# helps to control. So the numbers of w become smaller
XTX = [
    [1.0001, 2, 2],
    [2, 1.0001, 1.0000001],
    [2, 1.0000001, 1.0001]
]
```

```
XTX = np.array(XTX)
np.linalg.inv(XTX)
# Output:
# array([[-3.33366691e-01,  3.33350007e-01,  3.33350007e-01],
#             [ 3.33350007e-01,  5.00492166e+03, -5.00508835e+03],
#             [ 3.33350007e-01, -5.00508835e+03,  5.00492166e+03]])
```

The larger the number alpha adding to the diagonal, the more we have these weights under control. The reason why this works this way is, that this decrease the likelihood that these two columns are just copies of each other.
Alpha = 0.01

```
XTX = [
    [1.01, 2, 2],
    [2, 1.01, 1.0000001],
    [2, 1.0000001, 1.01]
]

XTX = np.array(XTX)
np.linalg.inv(XTX)
# Output:
# array([[ -0.33668908,   0.33501399,   0.33501399],
#             [  0.33501399,  49.91590897, -50.08509104],
#             [  0.33501399, -50.08509104,  49.91590897]])
```

To add alpha, we can use Identity matrix, multiply it with alpha value and add that matrix to XTX

```
XTX = XTX + 0.01 * np.eye(3)
```

We can keep the alpha as variable instead of hard-coding so we can tweak it as per our needs

We define new function regularized version train_linear_regression_reg(X, y , r=0.001)

```
# reg = regularized
# parameter r = short for regularization
def train_linear_regression_reg(X, y, r=0.001):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    XTX = XTX + r*np.eye(XTX.shape[0])

    XTX_inv = np.linalg.inv(XTX)
    w_full = XTX_inv.dot(X.T).dot(y)

    return w_full[0], w_full[1:]
```

And use it to train the model

```
X_train = prepare_X(df_train)
w0, w = train_linear_regression_reg(X_train, y_train, r=0.01)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)
rmse(y_val, y_pred)
```
✓ 0.2s

```
0.4608208286178567
```

RMSE improved from very high value to 0.46 which is better than previous value of 0.507
Now, if we increase alpha, r to 10, the RMSE = 0.4726 and if r = 1000, RMSE = 0.966 i.e. performance degrades
So, how do we determine best value of r ??

Tuning the model (Determining Regularization Alpha)

For different values of r, compare the RMSE

```
## For different values of r, compare the RMSE
for r in [0.0, 0.00001, 0.0001, 0.001, 0.1, 1, 10]:
    X_train = prepare_X(df_train)
    w0, w = train_linear_regression_reg(X_train, y_train, r=r)

    X_val = prepare_X(df_val)
    y_pred = w0 + X_val.dot(w)
    score = rmse(y_val, y_pred)

    print(r, w0, score)
```

```
0.0 -676260605536135.5 416.8831235468104
1e-05 8.221293309009221 0.4608153185904618
0.0001 7.134670083725352 0.4608153645357055
0.001 7.1308753574562935 0.46081585826182075
0.1 7.000232412717805 0.4608736549101277
1 6.250747847357639 0.4615812838276859
10 4.72951258571979 0.4726098772665481
```

We see that score is least for r = 0.001. So, we set that for our model

```
r = 0.001
X_train = prepare_X(df_train)
w0, w = train_linear_regression_reg(X_train, y_train, r=r)

X_val = prepare_X(df_val)
y_pred = w0 + X_val.dot(w)
score = rmse(y_val, y_pred)
score
```

Linear Regression with Regularization alpha is also called as **RIDGE REGRESSION**

Using the model

Till now, we found the best parameter for the linear regression and we trained our model on training dataset and applied the best model on validation dataset.
Now, we train our final model on both training dataset and validation dataset. We call this FULL TRAIN. After that we make the final evaluation on the test dataset to make sure that our model works fine and check what is the value for RMSE (how far/close it is with RMSE of validation dataset)

Combine the datasets

Combine df_train and df_val into one dataset. We can use Pandas concat() function that takes a list of dataframes and concatenates them together.

```
# Combine Train+Val = Full train
df_full_train = pd.concat([df_train, df_val])
df_full_train
```

✓ 0.0s

	make	model	year	
0	chevrolet	cobalt	2008	
1	toyota	matrix	2012	
2	subaru	impreza	2016	
3	volkswagen	vanagon	1991	
4	ford	f-150	2017	flex-f
...	...	...	...	
2377	volvo	v60	2015	
2378	maserati	granturismo_convertible	2015	premium_u
2379	cadillac	escalade_hybrid	2013	
2380	mitsubishi	lancer	2016	
2381	kia	sorento	2015	

9532 rows × 15 columns

Resetting index

In here, the index of result df_full_train is set as per indexes of individual dataframe. So, we drop and reset the index

```
# Drop and reset index of df_full_train
df_full_train = df_full_train.reset_index(drop=True)
```

Train the final model

Prepare the full_train X and get the full_train y vector using numpy.concatenate()

```
# Prepare the full_train X
X_full_train = prepare_X(df_full_train)
X_full_train
```

✓ 0.1s

```
array([[148.,   4.,  33., ...,   1.,   0.,   0.],
       [132.,   4.,  32., ...,   0.,   0.,   1.],
       [148.,   4.,  37., ...,   0.,   0.,   1.],
       ...,
       [332.,   8.,  23., ...,   0.,   0.,   0.],
       [148.,   4.,  34., ...,   0.,   0.,   0.],
       [290.,   6.,  25., ...,   0.,   0.,   0.]])
```

```
# Get the full_train y
y_full_train = np.concatenate([y_train, y_val])
```

Train the model on Full Train X i.e. get the w0, w

```
w0, w = train_linear_regression_reg(X_full_train, y_full_train, r=0.001)
w0, w
```
✓ 0.0s

```
(7.175198340062636,
 array([ 1.80193323e-03,  1.26575152e-01, -6.78607822e-03,  7.75903908e-03,
        -5.33226034e-05, -9.73063831e-02, -1.27241584e+00, -1.30599152e+00,
        -9.95969172e-01, -6.27479241e-02,  1.82139246e-01,  2.83073967e-02,
         1.29063817e-02, -1.32624475e-01, -2.69110368e-01, -6.50985199e-01,
        -3.21721814e-01, -3.55424919e-01, -3.51216047e-01, -6.61059055e-01,
```

Apply model on Test data

Now, we input the X-test to make predictions, get its RMSE and compare RMSE with RMSE of Validation set

```
X_test = prepare_X(df_test)
y_pred = w0 + X_test.dot(w)
score = rmse(y_test, y_pred)
score
```
✓ 0.0s

```
0.4600753970034731
```

RMSE is very close. So, it means our model has learnt to generalize and predict against unseen data (unseen means the data that the model has not been trained on) 😊

Real-life use of model

So, now that we have the final model i.e. we have the w values, we can use this model to predict price of a car. Using the model means:

1. Extracting all the features (getting feature vector of the car)
2. Applying our final model to this feature vector & predicting the price

Feature Extraction

So, lets take any row from the Test dataframe. In real scenarios, we would get the data from the user via a frontend which will call our model through an API. So, the data that user has entered will come as a JSON dictionary.

For demo, we convert the datarow to Dictionary first

```
car = df_test.iloc[20].to_dict()
car
```
✓ 0.0s

```
{'make': 'toyota',
 'model': 'sienna',
 'year': 2015,
 'engine_fuel_type': 'regular_unleaded',
 'engine_hp': 266.0,
 'engine_cylinders': 6.0,
 'transmission_type': 'automatic',
 'driven_wheels': 'front_wheel_drive',
 'number_of_doors': 4.0,
 'market_category': nan,
 'vehicle_size': 'large',
 'vehicle_style': 'passenger_minivan',
 'highway_mpg': 25,
 'city_mpg': 18,
 'popularity': 2031}
```

```
## Convert that List of Dictionary to Dataframe
df_small = pd.DataFrame([car])
df_small
```

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders
0	toyota	sienna	2015	regular_unleaded	266.0	6.0

We use this single row DataFrame as input for the prepare_X() function to get the feature matrix. In this case our feature matrix is a feature vector.

```
X_small = prepare_X(df_small)
X_small
```
✓ 0.0s

```
array([[2.660e+02, 6.000e+00, 2.500e+01, 1.800e+01, 2.031e+03, 2.000e+00,
        0.000e+00, 0.000e+00, 1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
        1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
```

Predict the price

The final step is to apply the final model to our requested car (feature vector) and predict the price. y-pred is log value. So, to get real predicted value, do exponent. And, then see the actual value of the car from test dataset

```
y_pred = w0 + X_small.dot(w)
y_pred = y_pred[0]
y_pred
```
✓ 0.0s

```
10.63249250527425
```

```
np.expm1(y_pred)
```
✓ 0.0s

```
41459.3366269011
```

```
np.expm1(y_test[20])
```
✓ 0.0s

```
35000.00000000001
```

3. BINARY CLASSIFICATION: CHURN PREDICTION PROJECT

Assume we have a Telecom company TELCO1 with many subscribers. Some of the subscribers are unhappy with the services of TELCO1 and are planning to move to TELCO2. TELCO1 wants to be able to predict which users will "churn out". For those likely to churn, TELOC1 will send them emails with some discounts so that they stay. Here, correctly predicting the likely users is important as we do not want to give discount to user who is not going to churn → that will be a loss for TELCO1

So, this is an example of Binary classification where the output is 0 (Not churn) or 1 (Will churn). Model will output values between 0 and 1 which are the probabilities of churning for 5 users could be 20%, 35%, 40%, 50%, 85% and we need to classify them as either 0 or 1.

Now, how does this model work? We use last month's data of all the subscribers - For subscribers who left, assign 1 and for those who stayed, assign 0 and this becomes the target vector y. The Feature matrix X would be information about the user like demographics, where did they stay, which services they use etc.

We use dataset from Kaggle - https://www.kaggle.com/datasets/blastchar/telco-customer-churn

In this project, we will use libraries from scikit-learn for preparing the data, classification etc instead of building our own.

Data Preparation

We make the column names and categorical values as lowercase and replace whitespaces with underscores. We also rectify the datatypes of columns like 'Seniorcitizen' was represented as a numerical value (0 or 1) rather than a string ('yes' or 'no'), and 'totalcharges' is currently classified as an object(string) but should be a numerical data type. Also, the churn values are 'yes' or 'no' but we need it to be numeric for machine learning. So, we also rectify it

Refer to Jupyter notebook

Setting up Validation Framework

In previous Car Price Prediction, we manually split the original dataset into Training, Validation & Test datasets. Here, we Perform the train/validation/test split with Scikit-Learn

```
# Import the train_test_split
from sklearn.model_selection import train_test_split
```

```
# See documentation of the function
train_test_split?
```

✓ 0.0s

```
Signature:
train_test_split(
    *arrays,
    test_size=None,
    train_size=None,
    random_state=None,
    shuffle=True,
    stratify=None,
)
Docstring:
Split arrays or matrices into random train and t
```

train_test_split() functions splits into 2 parts. So, to get our 3 parts of 60,20,20%, we first split original dataframe into 80%,20%. The 80% would be our FULL TRAIN Df and 20% would be df_test. Then, we again use train_test_split()

on df_fulltrain to break it into 75%, 25% df_train, df_val -> With this df_train will be 60% of df and df_val will be 20% of df

```
# Use train_test_split two times. We add random_state for reproducibility.
# First, we use test_size=20% on df to get df_full_train as 80%, df_test as 20%
# Then, we use test_size=25% on df_full_train to get df_train as 80%, df_val as 20%
df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=1)
df_train, df_val = train_test_split(df_full_train, test_size=0.25, random_state=1)
```

```
len(df_train), len(df_val), len(df_test)
```
✓ 0.0s

```
(4225, 1409, 1409)
```

train_test_split() function also shuffles the data and so the indexes in the df_* are out of order. Though it does not make any difference to the ML model, we drop and reset it for human readability

```
## Drop and Reset the indexes
df_train = df_train.reset_index(drop=True)
df_val = df_val.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)
```

```
## Set the Target vector arrays
y_train = df_train.churn.values
y_val = df_val.churn.values
y_test = df_test.churn.values
```

```
## Remove the target variable column
# from train, val, test df to
# avoid accidental usage
del df_train['churn']
del df_val['churn']
del df_test['churn']
```

Exploratory Data Analysis EDA

We'll cover

1. Checking missing values
2. Looking at the target variable (churn)
3. Looking at numerical and categorical variables

Check missing values

```
df_full_train.isnull().sum()

# Output:
# customerid          0
# gender              0
# seniorcitizen       0
# partner             0
# dependents          0
# tenure              0
# phoneservice        0
# multiplelines       0
# internetservice     0
# onlinesecurity      0
# onlinebackup        0
# deviceprotection    0
# techsupport         0
# streamingtv         0
# streamingmovies     0
# contract            0
# paperlessbilling    0
# paymentmethod       0
# monthlycharges      0
# totalcharges        0
# churn               0
# dtype: int64
```

There are no missing values in df_full_train. so, no action required

Look at the target variable (churn)

```
df_full_train.churn

# Output:
# 0       0
# 1       1
# 2       0
# 3       0
# 4       0
#        ..
# 5629    1
# 5630    0
# 5631    1
# 5632    1
# 5633    0
# Name: churn, Length: 5634,
```

churn is now an int having only 0 and 1s.

Check the distribution of our target variable 'churn'. How many customers are churning and how many are not-churning.

```
df_full_train.churn.value_counts()

# Output:
# 0    4113
# 1    1521
# Name: churn, dtype: int64
```

```
df_full_train.churn.value_counts(normalize=True)

# Output:
# 0    0.730032
# 1    0.269968
# Name: churn, dtype: float64
```

There is a total of 5634 customers. Among these, 1521 are dissatisfied customers (churning), while the remaining 4113 are satisfied customers (not churning). Understanding the distribution of your target variable is an essential step in any data analysis or modeling task, as it provides valuable insights into the data's class balance, which can influence modeling decisions and evaluation metrics.

Using the value_counts() function with the normalize=True parameter provides percentage. So, the churn rate, which represents the proportion of churning customers relative to the total number of customers. In our case, we've calculated that the churn rate is almost 27%.

Since "churn" has only two values 0 and 1, the churn rate can also be calculated using df_full_train.churn.mean() which also gives 27%

```
global_churn_rate = df_full_train.churn.mean()
round(global_churn_rate, 2)

# Output: 0.27
```

Explanation- The formula for the mean is given by:

$$mean=(1/n)\sum x$$

In this case, where $x \in \{0,1\}$, it simplifies to:

$$mean=(number\ of\ ones)/n$$

which is equal to value_counts(normalize=true).
This principle holds true for all binary datasets, because the mean of binary values corresponds directly to the proportion of ones in the dataset, which is essentially the churn rate in this context.

Look at the numerical, categorical variables

```
# Look for numerical, categorical columns
df_full_train.dtypes
```

✓ 0.0s

```
customerid          object
gender              object
seniorcitizen        int64
partner             object
dependents          object
tenure               int64
phoneservice        object
multiplelines       object
internetservice     object
onlinesecurity      object
onlinebackup        object
deviceprotection    object
techsupport         object
streamingtv         object
streamingmovies     object
contract            object
paperlessbilling    object
paymentmethod       object
monthlycharges     float64
totalcharges       float64
churn                int32
```

There are 3 numerical columns - tenure, monthlycharges, totalcharges
All other columns except customerid, churn are categorical columns

```
numerical = ['tenure', 'monthlycharges', 'totalcharges']

# Removing 'customerid', 'tenure', 'monthlycharges', 'totalcharges', and 'ch
categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
       'phoneservice', 'multiplelines', 'internetservice',
       'onlinesecurity', 'onlinebackup', 'deviceprotection', 'techsupport',
       'streamingtv', 'streamingmovies', 'contract', 'paperlessbilling',
       'paymentmethod']
```

We see the distribution of categorical columns through no. of unique

```
# Look at the no. of unique values for categories
df_full_train[categorical].nunique()
```

✓ 0.0s

```
gender              2
seniorcitizen       2
partner             2
dependents          2
phoneservice        2
multiplelines       3
internetservice     3
onlinesecurity      3
onlinebackup        3
deviceprotection    3
techsupport         3
streamingtv         3
streamingmovies     3
contract            3
paperlessbilling    2
paymentmethod       4
```

Feature Importance: Churn Rate, Risk Ratio

Feature importance analysis is a part of exploratory data analysis (EDA) and involves identifying which features affect our target variable. In this section, lets focus on two metrics - Churn Rate & Risk Ratio

Churn Rate

It is the percentage of no. of people who have churned out of the total no of people. We saw the Global Churn Rate = 27%. Now, we are focusing on the churn rate within different groups.

For example, we are interested in determining the churn rate for the gender group. We can calculate the churn rates for the female and male groups.

```
global_churn = df_full_train.churn.mean()
global_churn
# Output: 0.26996805111821087

churn_female = df_full_train[df_full_train.gender == 'female'].churn.mean()
churn_female
# Output: 0.27682403433476394

global_churn - churn_female
# Output: -0.006855983216553063

churn_male = df_full_train[df_full_train.gender == 'male'].churn.mean()
churn_male
# Output: 0.2632135306553911

global_churn - churn_male
# Output: 0.006754520462819769
```

We observe that the female churn rate is slightly higher than the global rate, while the male churn rate is slightly lower than the global rate. This suggests that women are somewhat more likely to churn.

Let’s check the churn rate of another group (with partner vs. without partner).

```
df_full_train.partner.value_counts()

# Output:
# no     2932
# yes    2702
# Name: partner, dtype: int64
```

```
global_churn = df_full_train.churn.mean()
global_churn
# Output: 0.26996805111821087

churn_partner = df_full_train[df_full_train.partner == 'yes'].churn.mean()
churn_partner
# Output: 0.20503330866025166

global_churn - churn_partner
# Output: 0.06493474245795922

churn_no_partner = df_full_train[df_full_train.partner == 'no'].churn.mean()
churn_no_partner
# Output: 0.3298090040927694

global_churn - churn_no_partner
# Output: -0.05984095297455855
```

When examining this group, we notice that customers with partners are significantly less likely to churn. The churn rate for this group is approximately 20.5%, contrasting with the global churn rate of almost 27%. On the other hand, customers without partners have a much higher churn rate compared to the global rate, standing at 33% as opposed to 27%.
This observation suggests that the partner variable may be more influential for predicting churn than the gender variable.

Risk Ratio

In the context of machine learning and classification, the "risk ratio" typically refers to a statistical measure used to assess the likelihood or probability of a certain event occurring in one group compared to another. It's a useful concept in various fields, including healthcare, finance, and customer churn analysis.

In the specific context of churn rate, the risk ratio can help you understand the relative risk of churn (i.e., customers leaving) for different groups or segments within your dataset. It can provide insights into which features or factors are associated with a higher or lower risk of churn.

Here's a simplified explanation of how risk ratio works in the context of churn rate:

1. **Definition of Risk Ratio:** The risk ratio (also known as the relative risk) is defined as the probability of an event occurring in one group divided by the probability of the same event occurring in another group. In the case of churn rate, you're typically comparing two groups: one group that exhibits a certain characteristic or behavior (e.g., customer has churned) and another group that does not exhibit that characteristic (e.g., customer hasn't churned).

2. **Interpretation:** A risk ratio greater than 1 suggests that the event (churn in this case) is more likely in the first group compared to the second group. A risk ratio less than 1 suggests the event is less likely in the first group. A risk ratio equal to 1 means there is no difference in risk between the two groups.

3. **Application:** We can use risk ratios to assess the impact of different features or interventions on churn rate. For example, we might calculate the risk ratio of churn for customers who received a promotional offer versus those who did not. If the risk ratio is significantly greater than 1, it indicates that the promotional offer had a positive impact on reducing churn.

4. **Statistical Significance:** It's important to also consider statistical significance when interpreting risk ratios. Statistical tests such as chi-squared tests or confidence intervals can help determine if the observed differences in churn rates are statistically significant.

So the risk ratio is a valuable tool for assessing the impact of different factors or features on churn rate in classification tasks. It helps you quantify and compare the relative risk of churn between different groups, providing insights that can inform decision-making and strategies for reducing churn.

Let's compare the risk ratio for churning between people with partners and those without partners.

```
churn_no_partner / global_churn
# Output: 1.2216593879412643

churn_partner / global_churn
# Output: 0.7594724924338315
```

This demonstrates that the churn rate for people without partners is 22% higher, whereas for people with partners, it is 24% lower than the global churn rate.

Let's take the data and group it by gender, and for each variable within the gender group, let's calculate the average churn rate within that group and calculate the difference and risk. We can perform this analysis for all the variables, not just the gender variable. The SQL query would look like:

```
SELECT gender,
AVG(churn),
AVG(churn) - global_churn AS diff,
AVG(churn) / global_churn AS risk
FROM data
GROUP BY gender;
```

In Pandas groupby() function, we can directly do the mean like in SQL but we need to manually add the difference calculations. .mean() returns a Pandas series. But, if we want a Dataframe, we can use the .agg([list of aggregate functions]) function which computes aggregate functions as per input list and outputs a Dataframe. So, we take mean, count in the aggregate function

```
df_full_train.groupby('gender').churn.mean()

# Output:
# gender
# female    0.276824
# male      0.263214
# Name: churn, dtype: float64
```

```
# agg takes a list of different aggregations
df_full_train.groupby('gender').churn.agg(['mean', 'count'])
```

gender	mean	count
female	0.276824	2796
male	0.263214	2838

And then add "diff", "risk" columns to the dataframe with the calculations

```
df_group = df_full_train.groupby('gender').churn.agg(['mean', 'count'])
df_group['diff'] = df_group['mean'] - global_churn
df_group['risk'] = df_group['mean'] / global_churn
df_group
```

gender	mean	count	diff	risk
female	0.276824	2796	0.006856	1.025396
male	0.263214	2838	-0.006755	0.974980

Now, lets get such stats for all the categorical columns. For this, we add the above groupby.agg() logic in a for loop. And to display each dataframe in the Jupyter notebook one after another, we import display() from IPython.display module

```
from IPython.display import display

for c in categorical:
    #print(c)
    df_group = df_full_train.groupby(c).churn.agg(['mean', 'count'])
    df_group['diff'] = df_group['mean'] - global_churn
    df_group['risk'] = df_group['mean'] / global_churn
    display(df_group)
    print()
    print()
```

gender	mean	count	diff	risk
female	0.276824	2796	0.006856	1.025396
male	0.263214	2838	-0.006755	0.974980

seniorcitizen	mean	count	diff	risk
0	0.242270	4722	-0.027698	0.897403
1	0.413377	912	0.143409	1.531208

partner	mean	count	diff	risk
no	0.329809	2932	0.059841	1.221659
yes	0.205033	2702	-0.064935	0.759472

dependents	mean	count	diff	risk
no	0.313760	3968	0.043792	1.162212
yes	0.165666	1666	-0.104302	0.613651

phoneservice	mean	count	diff	risk
no	0.241316	547	-0.028652	0.893870
yes	0.273049	5087	0.003081	1.011412

multiplelines	mean	count	diff	risk
no	0.257407	2700	-0.012561	0.953474
no_phone_service	0.241316	547	-0.028652	0.893870
yes	0.290742	2387	0.020773	1.076948

internetservice	mean	count	diff	risk
dsl	0.192347	1934	-0.077621	0.712482
fiber_optic	0.425171	2479	0.155203	1.574895
no	0.077805	1221	-0.192163	0.288201

onlinesecurity	mean	count	diff	risk
no	0.420921	2801	0.150953	1.559152
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.153226	1612	-0.116742	0.567570

onlinebackup	mean	count	diff	risk
no	0.404323	2498	0.134355	1.497672
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.217232	1915	-0.052736	0.804660

deviceprotection	mean	count	diff	risk
no	0.395875	2473	0.125907	1.466379
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.230412	1940	-0.039556	0.853480

techsupport	mean	count	diff	risk
no	0.418914	2781	0.148946	1.551717
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.159926	1632	-0.110042	0.592390

streamingtv	mean	count	diff	risk
no	0.342832	2246	0.072864	1.269897
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.302723	2167	0.032755	1.121328

streamingmovies	mean	count	diff	risk
no	0.338906	2213	0.068938	1.255358
no_internet_service	0.077805	1221	-0.192163	0.288201
yes	0.307273	2200	0.037305	1.138182

contract	mean	count	diff	risk
month-to-month	0.431701	3104	0.161733	1.599082
one_year	0.120573	1186	-0.149395	0.446621
two_year	0.028274	1344	-0.241694	0.104730

paperlessbilling	mean	count	diff	risk
no	0.172071	2313	-0.097897	0.637375
yes	0.338151	3321	0.068183	1.252560

paymentmethod	mean	count	diff	risk
bank_transfer_(automatic)	0.168171	1219	-0.101797	0.622928
credit_card_(automatic)	0.164339	1217	-0.105630	0.608733
electronic_check	0.455890	1893	0.185922	1.688682
mailed_check	0.193870	1305	-0.076098	0.718121

Values greater than 1 suggest a higher likelihood to churn, whereas values less than 1 suggest a lower likelihood to churn. We observe certain categories in which people tend to churn more or less frequently compared to the global average. These are the types of variables we are interested in and want to use in machine learning algorithms.
For example, “Gender” has risk ratios approx. 1 so that variable is not significant. But a SeniorCitizen has risk ratio >1.5 and is more likely to churn. High Risk Ratio is also seen in group without partner and with dependents.

While it’s informative to see this for individual variables in each table, it would be valuable to have a measure that quantifies the overall importance of each variable so that we can determine whether the “contract” variable is less or more important than “streamingmovies” and so on

Measuring Feature Importance for Categorical Features: Mutual information

Risk Ratio provides valuable insights into the importance of different categorical variables, particularly when examining the likelihood of churn for each value within a variable. For example, when analyzing the “contract” variable with values like “month-to-month,” “one_year,” and “two_years,” we can observe that customers with a “month-to-month” contract are more likely to churn compared to those with a “two_years” contract. This suggests

that the “contract” variable is likely to be an important factor in predicting churn. However, without a way to compare this importance with other variables, we may not have a clear understanding of its relative significance.

Mutual information, a concept from information theory, addresses this issue by quantifying how much we can learn about one variable when we know the value of another. The higher the mutual information, the more information we gain about churn by observing the value of another variable. In essence, it provides a means to measure the importance of categorical variables and their values in predicting churn, allowing us to compare their significance relative to one another.

To calculate Mutual information score, we use scikit learn function mutual_info_score(). It takes two arguments i.e. the 2 variables between which we need to compute the score

```
from sklearn.metrics import mutual_info_score

mutual_info_score(df_full_train.churn, df_full_train.contract)
# order is not important
#mutual_info_score(df_full_train.contract, df_full_train.churn)
# Output: 0.0983203874041556

mutual_info_score(df_full_train.churn, df_full_train.gender)
# Output: 0.0001174846211139946

mutual_info_score(df_full_train.churn, df_full_train.partner)
# Output: 0.009967689095399745
```

We see that MIS between Gender-Churn is very low (0.0001) while between Contract-Churn is high(0.098). It means that if we know the Gender of any customer, we cannot satisfactorily predict the Churn likelihood, but if we know the Contract of a customer, we can more confidently predict the Churn likelihood. Also, in other words, for Machine Learning, we understand that “Contract” is a much more relatively important feature to consider than “Gender” to predict Churn Likelihood

Now, lets calculate MIS between Churn and all the categorical features. For this we use the dataframe.apply() function to evaluate MIS for each column. But, apply() can take only on argument while MIS function takes 2 arguments. So, we wrap the MIS function into a wrapper function mutual_info_churn_score(series) which calls the MIS function with series as variable argument and df.churn as hardcoded argument. Then, we call .apply(mutual_info_churn_score) to the categorical columns

```
def mutual_info_churn_score(series):
    return mutual_info_score(series, df_full_train.churn)

mi = df_full_train[categorical].apply(mutual_info_churn_score)
mi

# Output:
# gender              0.000117
# seniorcitizen       0.009410
# partner             0.009968
# dependents          0.012346
# phoneservice        0.000229
# multiplelines       0.000857
# internetservice     0.055868
# onlinesecurity      0.063085
# onlinebackup        0.046923
# deviceprotection    0.043453
# techsupport         0.061032
# streamingtv         0.031853
# streamingmovies     0.031581
# contract            0.098320
# paperlessbilling    0.017589
# paymentmethod       0.043210
# dtype: float64
```

```
mi.sort_values(ascending=False)

# Output:
# contract            0.098320
# onlinesecurity      0.063085
# techsupport         0.061032
# internetservice     0.055868
# onlinebackup        0.046923
# deviceprotection    0.043453
# paymentmethod       0.043210
# streamingtv         0.031853
# streamingmovies     0.031581
# paperlessbilling    0.017589
# dependents          0.012346
# partner             0.009968
# seniorcitizen       0.009410
# multiplelines       0.000857
# phoneservice        0.000229
# gender              0.000117
# dtype: float64
```

To have the most important variables appear at the top of the list, we sort the variables based on their mutual information scores in descending order

Measuring Feature Importance for Numerical features: Correlation

For measuring feature importance for numerical variables, one common approach is to use the correlation coefficient, specifically Pearson's correlation coefficient. The Pearson correlation coefficient quantifies the degree of linear dependency between two numerical variables.

The correlation coefficient (often denoted as "r") has a range of -1 to 1:

- A negative correlation (r = -1) indicates a strong inverse relationship, where one variable increases as the other decreases.
- A positive correlation (r = 1) indicates a strong positive relationship, where both variables increase together.
- An r value close to 0 suggests a weak or no linear relationship between the variables.

The strength of the correlation is indicated by the absolute value of r:

- 0.0 < |r| < 0.2: Low correlation
- 0.2 < |r| < 0.5: Moderate correlation
- 0.6 < |r| < 1.0: Strong correlation

To calculate the Pearson correlation coefficient in Python, you can use the corr() function from pandas:
correlation = df[numerical_vars].corr()['churn']
This will give you a series of correlation values between each numerical variable and the churn variable, and you can sort them in descending order to identify the most important numerical features.

In the context of churn prediction:

- Positive Correlation: A positive correlation between a numerical variable (e.g., tenure) and churn means that as the numerical variable increases (e.g., longer tenure), the likelihood of churn (1) increases.
- Negative Correlation: A negative correlation between a numerical variable (e.g., tenure) and churn means that as the numerical variable increases (e.g., longer tenure), the likelihood of churn (1) decreases.
- Zero Correlation: A correlation close to zero suggests that there is no significant linear relationship between the numerical variable and churn.

```
df_full_train[numerical].corrwith(df_full_train.churn)
```

✓ 0.1s

```
tenure          -0.351885
monthlycharges   0.196805
totalcharges    -0.196353
```

From the above correlation data, since the correlation coefficient between tenure and churn is -ve, it means that as tenure increases, the likelihood of churn decreases. And, since the correlation coefficient between monthlycharges and churn is +ve, it means that as monthlycharges increase, the likelihood of churn also increases

```
df_full_train[df_full_train.tenure <= 2].churn.mean()
# Output: 0.5953420669577875

df_full_train[df_full_train.tenure > 2].churn.mean()
# Output: 0.22478269658378816

df_full_train[(df_full_train.tenure > 2) & (df_full_train.tenure <= 12)].chu
# Output: 0.3994413407821229

df_full_train[df_full_train.tenure > 12].churn.mean()
# Output: 0.17634908339788277
```

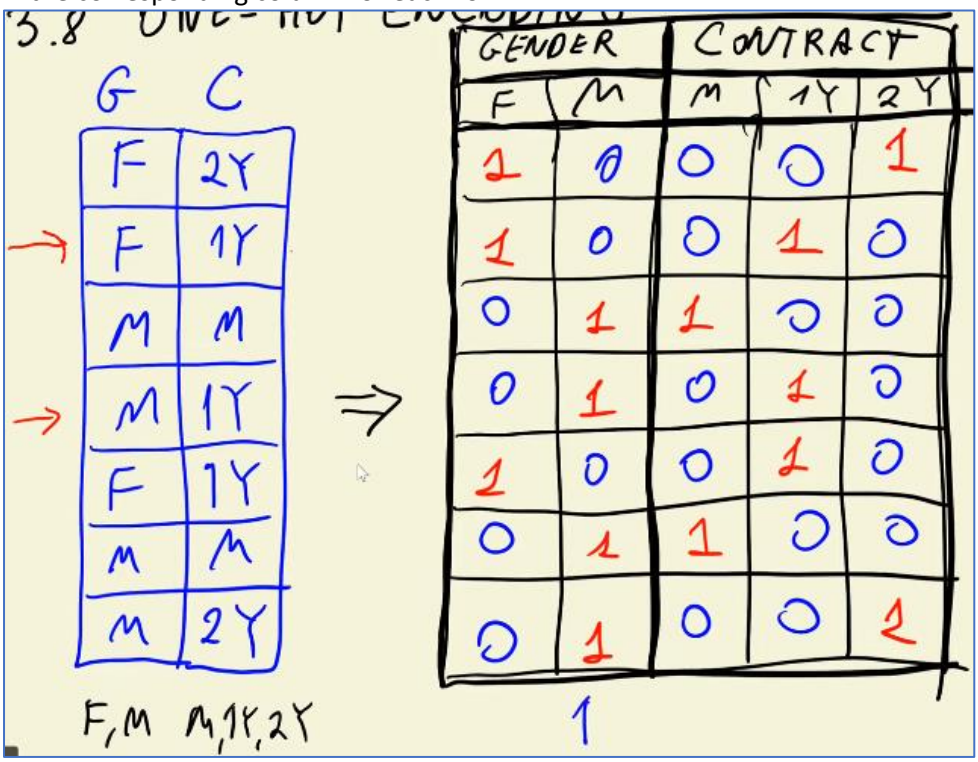
Regarding the variable “tenure,” we can observe that the group of customers with the highest likelihood to churn consists of those with a tenure of less than or equal to 2. It has a churn rate of almost 60%.

```
df_full_train[df_full_train.monthlycharges <= 20].churn.mean()
```
✓ 0.0s
```
0.08795411089866156
```
```
df_full_train[(df_full_train.monthlycharges > 20) & (df_full_train.monthlycharges <= 50)].churn.mean()
```
✓ 0.0s
```
0.18340943683409436
```
```
df_full_train[df_full_train.monthlycharges > 50].churn.mean()
```
✓ 0.0s
```
0.32499341585462205
```
Concerning the variable “monthlycharges,” we can observe that the group of customers with the highest likelihood to churn belongs to the group with monthly charges greater than $50. It has a churn rate of 32.5%.

One hot encoding

One-hot encoding is a technique used in machine learning to convert categorical (non-numeric) data into a numeric format that can be used by machine learning algorithms. In the previous car Price prediction, we did this encoding by manual logic. Here, we will use Scikit library for this.

To understand, consider our 2 categorical features, Gender (Male, Female) & Contract (Monthly, 1Year, 2Year). For Gender, we have two values while for Contract we have 3 values. So, we will add 2+3 = 5 new columns. And place 1 in the corresponding column for each row.



Use Scikit Learn for One Hot Encoding

There are many ways of implementing One Hot Encoding. One way is to use Scikit learn.DictVectorizer. See screenshots with comments below

```
## Import DictVectorizer
from sklearn.feature_extraction import DictVectorizer
```

```
## Small df,Take 1st 100 rows and gender,
# contract column of Full train
df_full_train[['gender','contract']].iloc[:100]
```

→

	gender	contract
0	male	two_year
1	female	one_year
2	male	two_year
3	male	one_year
4	male	one_year
...	...	...
95	male	two_year
96	male	two_year
97	male	month-to-month
98	male	month-to-month
99	male	month-to-month

100 rows × 2 columns

```
## Since DictVectorizer works on Dictionaries, convert small df to dict
dicts = df_full_train[['gender','contract']].iloc[:100].to_dict(orient='records')
```

```
## Initialize DictVectorizer
dv = DictVectorizer(sparse=False)
```

```
## Train the DV-Show the DV how the data looks like
# DV infers the columns and the values and
# learns the data to know what it has to do
dv.fit(dicts)
```

✓ 0.0s

```
DictVectorizer(sparse=False)
```

```
## Check the feature-names in the DV
dv.feature_names_
```

✓ 0.0s

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
 'gender=female',
 'gender=male']
```

```
## Then, we transform to do the one hot encoding.
# Since we set sparse=False during DV initialization,
# it returns a numpy array of 1 and 0
dv.transform(dicts)
```

✓ 0.0s

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

```
## DictVectorizer is smart enough to know if a feature is
# categorical or Numerical. So, if you mix categorical &
# numerical, it will only act on the categorical without
# modifying the numerical
dicts = df_full_train[['gender','contract', 'tenure']].iloc[:100].to_dict(orient='records')
```

```
dv.fit(dicts)
```

✓ 0.0s

```
DictVectorizer(sparse=False)
```

```
dv.feature_names_
```

✓ 0.0s

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
 'gender=female',
 'gender=male',
 'tenure']
```

```
dv.transform(dicts)
## Numerical features remain untouched
```

✓ 0.0s

```
array([[ 0.,  0.,  1.,  0.,  1., 12.],
       [ 0.,  1.,  0.,  1.,  0., 42.],
       [ 0.,  0.,  1.,  0.,  1., 71.],
       [ 0.,  1.,  0.,  0.,  1., 71.],
       [ 0.,  1.,  0.,  0.,  1., 30.],
```

```
### Having understood the above, we apply to
# all features in df_train
train_dicts = df_train[categorical + numerical].to_dict(orient='records')
```

```
train_dicts[0]
```
✓ 0.0s

```
{'gender': 'female',
 'seniorcitizen': 0,
 'partner': 'yes',
 'dependents': 'yes',
 'phoneservice': 'yes',
 'multiplelines': 'yes',
 'internetservice': 'fiber_optic',
 'onlinesecurity': 'yes',
 'onlinebackup': 'yes',
 'deviceprotection': 'yes',
```

```
## Fit DictVectorizer
dv.fit(train_dicts)
```
✓ 0.1s

```
DictVectorizer
DictVectorizer(sparse=False)
```

```
dv.feature_names_
```
✓ 0.0s

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
 'dependents=no',
 'dependents=yes',
```

```
### We can combine the fit, transform
# using dv.fit_transform()
# That is our Training Feature Matrix X_train
X_train = dv.fit_transform(train_dicts)
```

```
# Similarly, the Validation Feature Matrix Xval
# We only do dv.transform as dv.fit is already
# done when we extracted X_train
val_dicts = df_val[categorical + numerical].to_dict(orient='records')
X_val = dv.transform(val_dicts)
```

Logistic Regression for Binary Classification

Classification problems can be categorized into binary problems and multi-class problems. Binary problems are the types of problems that logistic regression is typically used to solve.

In linear regression, the target variable y can take any value from -infinity to +infinity
In logistic regression, the target variable y can only be 0 or 1.

Now, the equation for linear regression is -

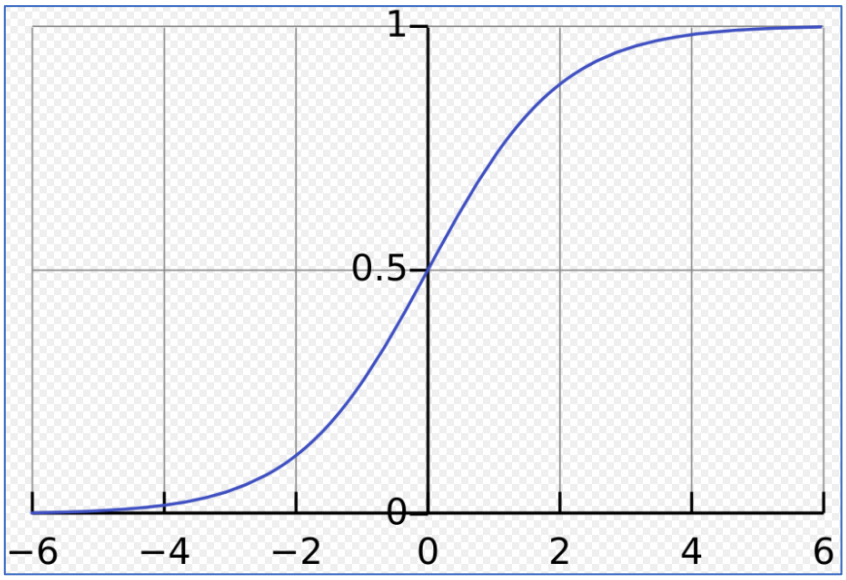
$$g(x_i) = w_0 + w^T x_i$$

Logistic regression uses the same equation with only additional SIGMOID function

$$g(x_i) = \text{SIGMOID}(w_0 + w^T x_i)$$

Sigmoid function

$$S(x) = \frac{1}{1 + e^{-x}}$$



This function maps any real number x to the range of 0 to 1, making it suitable for modeling probabilities in logistic regression. We'll use this function to convert a score into a probability.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

z = np.linspace(-7, 7, 51)
z
```

→

```
sigmoid(z)

# Output:
# array([9.11051194e-04, 1.20508423e-03, 1.59386223e-03, 2.10780106e-03,
#        2.78699622e-03, 3.68423990e-03, 4.86893124e-03, 6.43210847e-0
#        8.49286285e-03, 1.12064063e-02, 1.47740317e-02, 1.94550846e-0
#        2.55807883e-02, 3.35692233e-02, 4.39398154e-02, 5.73241759e-0
#        7.44679452e-02, 9.62155417e-02, 1.23467048e-01, 1.57095469e-0
#        1.97816111e-01, 2.46011284e-01, 3.01534784e-01, 3.63547460e-0
#        4.30453776e-01, 5.00000000e-01, 5.69546224e-01, 6.36452540e-0
#        6.98465216e-01, 7.53988716e-01, 8.02183889e-01, 8.42904531e-0
#        8.76532952e-01, 9.03784458e-01, 9.25532055e-01, 9.42675824e-0
#        9.56060185e-01, 9.66430777e-01, 9.74419212e-01, 9.80544915e-0
#        9.85225968e-01, 9.88793594e-01, 9.91507137e-01, 9.93567892e-0
#        9.95131069e-01, 9.96315760e-01, 9.97213004e-01, 9.97892199e-0
#        9.98406138e-01, 9.98794916e-01, 9.99088949e-01])
```

The first snippet below demonstrates the familiar linear regression, while the second snippet illustrates logistic regression. There is only one difference between the two: in logistic regression, the sigmoid function is applied to the result of the linear regression to transform it into a probability value between 0 and 1.

```
def linear_regression(xi):
    result = w0

    for j in range(len(w)):
        result = result + xi[j] * w[j]

    return result
```

```
def logistic_regression(xi):
    score = w0

    for j in range(len(w)):
        score = score + xi[j] * w[j]

    result = sigmoid(score)
    return result
```

Linear regression and logistic regression are called **linear models**, because dot product in linear algebra is a linear operator. Linear models are fast to use, fast to train.

Training Logistic Regression with Scikit-learn

- Train a model with Scikit-Learn
- Apply model to the validation dataset
- Calculate the accuracy

Train a model with Scikit-Learn

```
#Import the Logistic Regression class from sklearn.linear_model module
from sklearn.linear_model import LogisticRegression
```

model.fit()

```
    # model = LogisticRegression(solver='lbfgs')
    # # solver='lbfgs' is the default solver in newer version of sklearn
    # # for older versions, you need to specify it explicitly

    # Instantiate Logistic Regression object
    # Fit the model using X_train, y_train
    model = LogisticRegression()
    model.fit(X_train, y_train)
```

✓ 0.1s

```
▾ LogisticRegression
LogisticRegression()
```

model.intercept_[0]

```
    ## Get the bias term, w0 using intercept_ property
    # Returns a 1-element Numpy array
    model.intercept_[0]
```

✓ 0.0s

```
-0.10900111836553165
```

model.coef_[0]

```
    ## Get the weights(co-efficents) w vector
    # Returns 2d Numpy array
    # 1st row is the w vector
    model.coef_[0].round(3)
```

✓ 0.0s

```
array([ 0.475, -0.175, -0.408, -0.03 , -0.078,  0.063, -0.089, -0.081,
       -0.034, -0.073, -0.335,  0.317, -0.089,  0.004, -0.258,  0.141,
        0.009,  0.063, -0.089, -0.081,  0.266, -0.089, -0.284, -0.231,
        0.124, -0.166,  0.058, -0.087, -0.032,  0.071, -0.059,  0.141,
       -0.249,  0.216, -0.12 , -0.089,  0.102, -0.071, -0.089,  0.052,
        0.213, -0.089, -0.232, -0.07 ,  0.   ])
```

model.predict()

```
    # Model training is Done.
    # Do a dummy prediction using X_train
    model.predict(X_train)
    ## This gives output as 0(no churn) & 1(churn)
    ## These are called "hard predictions"
```

✓ 0.0s

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

This provides **hard predictions**, meaning it assigns either zeros (representing “not churn”) or ones (representing “churn”). These hard predictions are called such because we already have the exact labels in the training data.

Apply the model

model.predict_proba() - Instead of hard predictions, we can generate **soft predictions** by using the predict_proba function

```
## Get the predicted probabilities
model.predict_proba(X_train)
# 1st column is probability of 0(no churn)
# 2nd column is probability of 1(no churn)
```

✓ 0.0s

```
array([[0.90445971, 0.09554029],
       [0.32064152, 0.67935848],
       [0.36632641, 0.63367359],
       ...,
       [0.4684833 , 0.5315167 ],
       [0.95746817, 0.04253183],
       [0.30121032, 0.69878968]])
```

```
## Apply the model to the Validation set
# Get the Predicted Probabilities of Churn(1)
# So, need only 2nd column
y_pred = model.predict_proba(X_val)[:, 1]
y_pred
```

✓ 0.0s

```
array([0.00898501, 0.20466232, 0.212384  , ..., 0
       0.83743516])
```

```
## Now, you can select your own threshold value
# below which it will be considered 0 (no churn)
# above which it will be considered 1 (no churn)
# We select 0.5 as threshold and
# generate boolean array
churn_decision = (y_pred >= 0.5)
churn_decision
```

✓ 0.0s

```
array([False, False, False, ..., False,  True,  True])
```

```
## Filter the df_val Where Churn_decision is true,
# These users are like ly to churn and so they
# can be sent discount email
df_val[churn_decision]
```

✓ 0.1s

	customerid	gender	seniorcitizen	partner	depen
3	8433-wxgna	male	0	no	
8	3440-jpscl	female	0	no	
11	2637-fkfsy	female	0	yes	

Calculate Accuracy

we'll use the accuracy metric instead of root mean squared error (RMSE). Accuracy is a common metric for evaluating classification models like logistic regression.

```
y_val
```

✓ 0.0s

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

```
churn_decision
```

✓ 0.0s

```
array([False, False, False, ...,
```

y_val is array of 0 and 1. churn_decision is array of False, True. You can directly compare y_val and churn_decision and Python is smart enough to consider 0 as False, 1 as True and it will give the result. In that comparison result, if you want to know the % of True, you can simply use the Mean() because it will do No of Ones/No of all = No of True/No of All. This is the accuracy = 80.3%

```
## What % of predictions are correct?
# Compare y_val and churn_decision values
# and take the mean (no. of True/No of all)
(y_val == churn_decision).mean()
## 80.34%
```

✓ 0.0s

```
0.8034066713981547
```

Interpretation of the Model

Now, our DictVectorizer (that we used for One hot encoding) has all the feature names and the model.coef_ has all the feature weights

```
dv.get_feature_names_out()
# Output:
# array(['contract=month-to-month', 'contract=one_year',
#        'contract=two_year', 'dependents=no', 'dependents=yes',
#        'deviceprotection=no', 'deviceprotection=no_internet_service'
#        'deviceprotection=yes', 'gender=female', 'gender=male',
#        'internetservice=dsl', 'internetservice=fiber_optic',
#        'internetservice=no', 'monthlycharges', 'multiplelines=no',
#        'multiplelines=no_phone_service', 'multiplelines=yes',
#        'onlinebackup=no', 'onlinebackup=no_internet_service',
#        'onlinebackup=yes', 'onlinesecurity=no',
#        'onlinesecurity=no_internet_service', 'onlinesecurity=yes',
#        'paperlessbilling=no', 'paperlessbilling=yes', 'partner=no',
#        'partner=yes', 'paymentmethod=bank_transfer_(automatic)',
#        'paymentmethod=credit_card_(automatic)',
#        'paymentmethod=electronic_check', 'paymentmethod=mailed_check
#        'phoneservice=no', 'phoneservice=yes', 'seniorcitizen',
#        'streamingmovies=no', 'streamingmovies=no_internet_service',
#        'streamingmovies=yes', 'streamingtv=no',
#        'streamingtv=no_internet_service', 'streamingtv=yes',
#        'techsupport=no', 'techsupport=no_internet_service',
#        'techsupport=yes', 'tenure', 'totalcharges'], dtype=object)
```

```
model.coef_[0].round(3)
# Output:
# array([ 0.475, -0.175, -0.408, -0.03 , -0.078,  0.063, -0.089, -0.0
#        -0.034, -0.073, -0.335,  0.316, -0.089,  0.004, -0.258,  0.14
#         0.009,  0.063, -0.089, -0.081,  0.266, -0.089, -0.284, -0.23
#         0.124, -0.166,  0.058, -0.087, -0.032,  0.07 , -0.059,  0.14
#        -0.249,  0.215, -0.12 , -0.089,  0.102, -0.071, -0.089,  0.05
#         0.213, -0.089, -0.232, -0.07 ,  0.   ])
```

To interpret what is the weight of each feature, we need to combine these two using zip() function. Assume we have a list of numbers and a string. Zip will pair each element from list a and string b. Zip returns an iterator, so it needs to be transformed into another form like list or dict. List will have tuples of a,b while Dict will have key-value pair a:b

```
a = [1, 2, 3, 4]
b = 'abcd'
```

,

```
list(zip(a,b))
```

✓ 0.0s

```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

,

```
dict(zip(a, b))
```

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

So, we combine our featurenames & weights using zip() function

```
dict(zip(dv.get_feature_names_out(), model.coef_[0].round(3)))
```

✓ 0.0s

```
{'contract=month-to-month': 0.475,
 'contract=one_year': -0.175,
 'contract=two_year': -0.408,
 'dependents=no': -0.03,
 'dependents=yes': -0.078,
 'deviceprotection=no': 0.063,
 'deviceprotection=no_internet_service': -0.089,
 'deviceprotection=yes': -0.081,
 'gender=female': -0.034,
 'gender=male': -0.073,
 'internetservice=dsl': -0.335,
 'internetservice=fiber_optic': 0.317,
 'internetservice=no': -0.089,
 'monthlycharges': 0.004,
 'multiplelines=no': -0.258,
 'multiplelines=no_phone_service': 0.141,
 'multiplelines=yes': 0.009,
 'onlinebackup=no': 0.063,
 'onlinebackup=no_internet_service': -0.089,
 'onlinebackup=yes': -0.081,
 'onlinesecurity=no': 0.266,
 'onlinesecurity=no_internet_service': -0.089,
 'onlinesecurity=yes': -0.284,
 'paperlessbilling=no': -0.231,
 'paperlessbilling=yes': 0.124,
...
 'techsupport=no': 0.213,
 'techsupport=no_internet_service': -0.089,
 'techsupport=yes': -0.232,
 'tenure': -0.07,
 'totalcharges': 0.0}
```

Features with positive weights will move the Sigmoid score towards 1 (likely to churn) while those with negative weights will move the Sigmoid score towards 0 (no churn)

Understand the interpretation

To understand this better, lets re-train our model on very small dataset of just 3 features - contract, tenure, monthlycharges and 10 records.

```
small = ['contract', 'tenure', 'monthlycharges']

df_train[small].iloc[:10]
```

```
df_train[small].iloc[:10].to_dict(orient='records')
# Output:
# [{'contract': 'two_year', 'tenure': 72, 'monthlycharges': 115.5},
# {'contract': 'month-to-month', 'tenure': 10, 'monthlycharges': 95.25},
# {'contract': 'month-to-month', 'tenure': 5, 'monthlycharges': 75.55},
# {'contract': 'month-to-month', 'tenure': 5, 'monthlycharges': 80.85},
# {'contract': 'two_year', 'tenure': 18, 'monthlycharges': 20.1},
# {'contract': 'month-to-month', 'tenure': 4, 'monthlycharges': 30.5},
# {'contract': 'month-to-month', 'tenure': 1, 'monthlycharges': 75.1},
# {'contract': 'month-to-month', 'tenure': 1, 'monthlycharges': 70.3},
# {'contract': 'two_year', 'tenure': 72, 'monthlycharges': 19.75},
# {'contract': 'month-to-month', 'tenure': 6, 'monthlycharges': 109.9}]
```

Take first 10 records and those 3 features from df_train and convert to dict → Small Training Set. Do the same from df_val → Small Validation Set.
Do one hot encoding using DictVectorizer and get Training Feature matrix X_train_small

```
dicts_train_small = df_train[small].to_dict(orient='records')
dicts_val_small = df_val[small].to_dict(orient='records')

dv_small = DictVectorizer(sparse=False)
dv_small.fit(dicts_train_small)

# three binary features for the contract variable and two numerical features
# monthlycharges and tenure
dv_small.get_feature_names_out()
# Output:
# array(['contract=month-to-month', 'contract=one_year',
#.             'contract=two_year', 'monthlycharges', 'tenure'], dtype=obje
```

```
X_train_small = dv_small.transform(dicts_train_small)
```

Perform LogisticRegression on x_train_small. Get intercept (bias term w0) and coefficients (weights)

```
model_small = LogisticRegression()
model_small.fit(X_train_small, y_train)

w0 = model_small.intercept_[0]
w0
# Output: -2.476775661122344

w = model_small.coef_[0]
w.round(3)
# Output: array([ 0.97 , -0.025, -0.949,  0.027, -0.036])
```

Combine the small featurenames and weights

```
dict(zip(dv_small.get_feature_names_out(), w.round(3)))

# Output:
# {'contract=month-to-month': 0.97,
#  'contract=one-year': -0.025,
#  'contract=two_year': -0.949,
#  'monthlycharges': 0.027,
#  'tenure': -0.036}
```

Score = w0 + SumProduct(Weight-FeatureValue)

Contract	Monthly charges	Tenure(months)	Score	Probability of Churn = Sigmoid(score)
Monthly	50	5	-2.47 + 0.97 + 50*0.027 + 5*(-0.036) = -0.33	0.41 (41%)
Monthly	60	1	-2.47 + 0.97 + 60*0.027 + 1*(-0.036) = 0.08399	0.5209 (52%)
2Y	30	24	-2.47 + (-0.949) + 30*0.027 + 24*(-0.036) = -3.473	0.03 (3%)

The above table shows the effect of Contract, Monthly Charges, Tenure on the Score and Churn Probability. As monthly charges increases Or Tenure Decreases, Churn probability increases.

Using the Model

Here, we will use the df_Full_train for Training and Predict on X_test and compare with y_val

Convert the Full Train df to Dictionary dicts_full_train

```
dicts_full_train = df_full_train[categorical + numerical].to_dict(orient='records')
dicts_full_train[:3]
```

✓ 0.1s

```
[{'gender': 'male',
  'seniorcitizen': 0,
  'partner': 'yes',
  'dependents': 'yes',
  'phoneservice': 'yes',
  'multiplelines': 'no',
  'internetservice': 'no',
  'onlinesecurity': 'no_internet_service',
  'onlinebackup': 'no_internet_service',
  'deviceprotection': 'no_internet_service',
  'techsupport': 'no_internet_service',
  'streamingtv': 'no_internet_service',
  'streamingmovies': 'no_internet_service',
  'contract': 'two_year',
  'paperlessbilling': 'no',
  'paymentmethod': 'mailed_check',
  'tenure': 12,
  'monthlycharges': 19.7,
  'totalcharges': 258.35},
 {'gender': 'female',
```

```
# create DictVectorizer
dv = DictVectorizer(sparse=False)

# from this dictionaries we get the feature matrix
X_full_train = dv.fit_transform(dicts_full_train)

# then we train a model on this feature matrix
y_full_train = df_full_train.churn.values
model = LogisticRegression()
model.fit(X_full_train, y_full_train)

# do the same things for test data
dicts_test = df_test[categorical + numerical].to_dict(orient='records')
X_test = dv.transform(dicts_test)

# do the predictions
y_pred = model.predict_proba(X_test)[:, 1]

# compute accuracy
churn_decision = (y_pred >= 0.5)
(churn_decision == y_test).mean()
# Output: 0.815471965933286
```

An accuracy of 81.5% on the test data is slightly more accurate than what we had in the validation data. Minor differences in performance are acceptable, but significant differences between training and validation/test data can indeed indicate potential issues with the model, such as overfitting. Ensuring that the model's performance is consistent across different datasets is an important aspect of model evaluation and generalization.

Let's imagine that we want to deploy the logistic regression model on a website where we can use it to predict whether a customer is likely to leave (churn) or not. When a customer visits the website and provides their information, this data is transferred as a dictionary over the network to the server hosting the model. The server then uses the model to compute a probability, which is returned to determine whether the customer is likely to

churn. This approach allows to make real-time predictions about customer churn and take appropriate actions, such as sending promotional offers to customers who are likely to leave.

Let's take a sample customer from our test set:

```
customer = dicts_test[10]
customer

# Output:
# {'gender': 'male',
#  'seniorcitizen': 1,
#  'partner': 'yes',
#  'dependents': 'yes',
#  'phoneservice': 'yes',
#  'multiplelines': 'no',
#  'internetservice': 'fiber_optic',
#  'onlinesecurity': 'no',
#  'onlinebackup': 'yes',
#  'deviceprotection': 'no',
#  'techsupport': 'no',
#  'streamingtv': 'yes',
#  'streamingmovies': 'yes',
#  'contract': 'month-to-month',
#  'paperlessbilling': 'yes',
#  'paymentmethod': 'mailed_check',
#  'tenure': 32,
#  'monthlycharges': 93.95,
#  'totalcharges': 2861.45}
```

To get the feature matrix for the requested customer as a dictionary, we create a list containing just that customer's dictionary.

```
X_small = dv.transform([customer])
X_small

# Output:
# array([[1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00,
#         1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00,
#         0.00000e+00, 1.00000e+00, 0.00000e+00, 9.39500e+01, 1.00000e+00,
#         0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00,
#         1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00,
#         0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00,
#         1.00000e+00, 0.00000e+00, 1.00000e+00, 1.00000e+00, 0.00000e+00,
#         0.00000e+00, 1.00000e+00, 0.00000e+00, 0.00000e+00, 1.00000e+00,
#         1.00000e+00, 0.00000e+00, 0.00000e+00, 3.20000e+01, 2.86145e+03]])

X_small.shape
# Output: (1, 45)
# one customer with 45 features
```

```
model.predict_proba(X_small)[0,1]

# Output: 0.4056810977975889
```

```
# Let's check the actuel value...
y_test[10]

# Output: 0
```

We see this customer has a probability of only 40% of churning i.e. we predict this customer is not going to churn and so don't send promotional email. And if we check the actual value in y_test, it is 0 meaning our decision to not send promotional email to this customer is correct.

Let's test one customer that is going to churn. This customer has a probability of almost 60% of churning and we predict this customer is going to churn so we send them promotional email). And if we check the actual value in y_test, it is 1 meaning our decision to send promotional email to this customer is correct.

```
customer = dicts_test[-1]
X_small = dv.transform([customer])
model.predict_proba(X_small)[0,1]

# Output: 0.5968852088398422
```

```
# Let's check the actuel value...
y_test[-1]

# Output: 1
```

4. EVALUATION METRICS

Background

Lets recap all the important lines of code that we used in the previous Churn Prediction Project. This includes the necessary imports, data preparation, data splitting for training, validation, and testing, separating the target variable 'churn', training the logistic regression model, and finally, validating the model on the validation data and outputting the accuracy at the end.

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

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import LogisticRegression
```

We used the necessary imports: Pandas, NumPy, and Matplotlib, as well as the three imports from the Scikit-Learn package. These imports are used once for the train-test-split, once for the DictVectorizer, and once for the linear model for LogisticRegression.

```
df = pd.read_csv('data-week-3.csv')

df.columns = df.columns.str.lower().str.replace(' ', '_')

categorical_columns = list(df.dtypes[df.dtypes == 'object'].index)

for c in categorical_columns:
    df[c] = df[c].str.lower().str.replace(' ', '_')

df.totalcharges = pd.to_numeric(df.totalcharges, errors='coerce')
df.totalcharges = df.totalcharges.fillna(0)

df.churn = (df.churn == 'yes').astype(int)
```

First, the dataset is read and stored in the 'df' dataframe. In the second line, the column names are standardized by converting them to lowercase and replacing spaces with underscores. Then, all categorical columns are assigned to the variable 'categorical_columns' using the condition 'dtypes == object'. However, it's worth noting that the 'totalcharges' column is mistakenly considered categorical, but it is, in fact, numerical. To correct this, the 'totalcharges' column is converted to a numerical format using the Pandas function 'to_numeric', with the 'errors='coerce" parameter set to ignore any errors. After this conversion, missing values are filled with '0'. Finally, the 'churn' column is converted to integer values, where 'churn==yes' becomes '1' and 'churn==no' becomes '0'.

```
df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=1)
df_train, df_val = train_test_split(df_full_train, test_size=0.25, random_state=1)

df_train = df_train.reset_index(drop=True)
df_val = df_val.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)

y_train = df_train.churn.values
y_val = df_val.churn.values
y_test = df_test.churn.values

del df_train['churn']
del df_val['churn']
del df_test['churn']
```

Next, we use the 'train_test_split' function to split the datasets into 'full_train' (80%) and 'test' in the first step. In the second step, 'full_train' is further divided into two datasets for training and validation. Ultimately, we achieve a split of the initial data into a 60%-20%-20% ratio, where 60% is used for training, and 20% each is allocated for validation and testing. The parameter 'random_state=1' ensures that the random split is reproducible. The next three lines reset the indices in each of the three datasets. Since the random split may result in non-continuous indices, setting 'drop=True' removes the old index. Then, for each record, the target variable 'y' is set, which in this case is the 'churn' column. Finally, the target column is removed from the records to prevent accidental usage during training.

```
numerical = ['tenure', 'monthlycharges', 'totalcharges']

categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
       'phoneservice', 'multiplelines', 'internetservice',
       'onlinesecurity', 'onlinebackup', 'deviceprotection', 'techsupport',
       'streamingtv', 'streamingmovies', 'contract', 'paperlessbilling',
       'paymentmethod']
```

We define two variables, 'numerical' and 'categorical', which contain the relevant column names. The 'numerical' array contains the names of all numerical columns, while the 'categorical' array contains the names of all categorical columns.

```
dv = DictVectorizer(sparse=False)

train_dict = df_train[categorical + numerical].to_dict(orient='records')
X_train = dv.fit_transform(train_dict)

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

Next, we create a DictVectorizer instance. We then transform the dataframe into dictionaries, and using the 'fit_transform(train_dict)' function, we train the DictVectorizer. This step involves showing the DictVectorizer how the data is structured, allowing it to distinguish column names and values and perform one-hot encoding based on this information. Importantly, the DictVectorizer is smart enough to distinguish between categorical values and numeric values, so numeric values are ignored during one-hot encoding. The 'transform' part of this process converts the dictionary into a vector or matrix suitable for machine learning. After preparing the data, we move on to model creation. In this case, a Logistic Regression model is used. The 'model.fit' function is then employed to train the model on the training data.

```
val_dict = df_val[categorical + numerical].to_dict(orient='records')
X_val = dv.transform(val_dict)

y_pred = model.predict_proba(X_val)[:, 1]
churn_decision = (y_pred >= 0.5)
(y_val == churn_decision).mean()

# Output: 0.8034066713981547
```

We validate the trained model using the validation data. To do this, we need to prepare the validation DataFrame in the same way as shown for the training DataFrame. This involves transforming it into dictionaries and applying the 'transform' function of the DictVectorizer. However, during validation, we only need to use the 'transform' function of the DictVectorizer since it already knows the data structure. In the case of validation, we are primarily interested in the transformed output, which serves as input for prediction. For prediction, we use the 'predict_proba' function of the model, which provides us with probabilities in two columns. Here, we are interested in the second column. We evaluate the model's performance using a threshold of '>=0.5'. The 'churn_decision' variable contains 'True' for any value in the prediction greater than or equal to the threshold, and 'False' otherwise. We calculate the accuracy using the 'mean' function, and in this case, it is approximately 80%.

ACCURACY & DUMMY MODEL

Accuracy measures the fraction of correct predictions made by the model.

In the Churn Prediction Project, we calculated that our model achieved an accuracy of 80% on the validation data. We checked each customer in the validation dataset to determine whether the model's churn prediction was correct or incorrect. This decision was based on our threshold of 0.5, meaning a customer with a predicted value greater than or equal to 0.5 was considered a churning customer, while values below the threshold were considered non-churning customers.

```
len(y_val)
# Output: 1409

(y_val == churn_decision).sum()
# Output: 1132

1132 / 1409
# Output: 0.8034

(y_val == churn_decision).mean()
# Output: 0.8034
```

Out of the 1409 customers in the validation dataset, we made 1132 correct predictions. Therefore, the accuracy is calculated as 1132/1409 = 0.80, which corresponds to 80%. This accuracy indicates that our model correctly predicted the churn status for 80% of the customers in the validation dataset. Whether this is considered good or not depends on the specific context and requirements of the problem.

Evaluating accuracy for different thresholds

We chose 0.5 as the threshold. What if we move the threshold to something like 0.3 or 0.7? Will the accuracy improve or degrade? So, we want to check if 0.5 is a good threshold or not. To evaluate this, we can adjust the threshold and perform validation again. By systematically varying the threshold, we can observe whether it improves the accuracy or not.

```
thresholds = np.linspace(0, 1, 21)
thresholds

# Output:
# array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 ,
#        0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95, 1.  ])
```

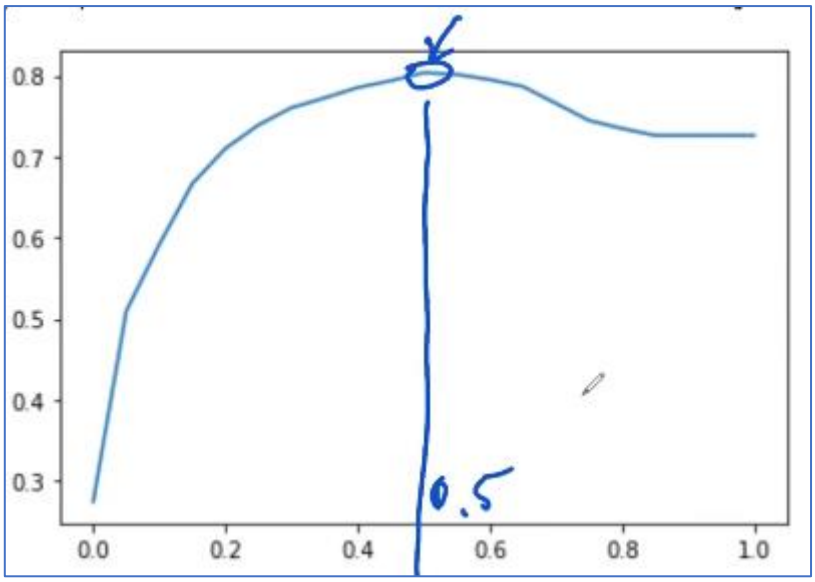
We use the 'linspace' function from NumPy to generate an array with multiple threshold values (e.g., 21 values evenly spaced between 0 and 1). For each threshold value, we can calculate the accuracy and then determine the best threshold value based on the validation results. This process allows us to fine-tune the threshold to optimize the model's performance.

```
scores = []

for t in thresholds:
    churn_decision = (y_pred >= t)
    score = (y_val == churn_decision).mean()
    print('%.2f %.3f' % (t, score))
    scores.append(score)
```

```
# Output:
# 0.00 0.274
# 0.05 0.509
# 0.10 0.591
# 0.15 0.666
# 0.20 0.710
# 0.25 0.739
# 0.30 0.760
# 0.35 0.772
# 0.40 0.785
# 0.45 0.793
# 0.50 0.803
# 0.55 0.801
# 0.60 0.795
# 0.65 0.786
# 0.70 0.766
# 0.75 0.744
# 0.80 0.735
# 0.85 0.726
# 0.90 0.726
# 0.95 0.726
# 1.00 0.726
```

For each threshold value, we set the threshold and calculate the accuracy score. The highest accuracy score of 80.3% happens with threshold 0.5. This suggests that the default threshold of 0.5 is an appropriate choice for our model in this context. To visually represent this threshold optimization process, we can create a plot. The x-axis will represent the threshold values, while the y-axis will represent the corresponding scores (in this case, accuracy or another relevant metric). This plot will provide a clear visualization of how the model's performance varies with different threshold values, helping us identify the threshold that maximizes the desired metric.

```
plt.plot(thresholds,scores)
```



Scikit Learn Function for Accuracy

We used our custom logic using .mean() to calculate accuracy. Scikit learn has built-in functions to measure accuracy and other metrics. Accuracy is in sklearn.metrics module

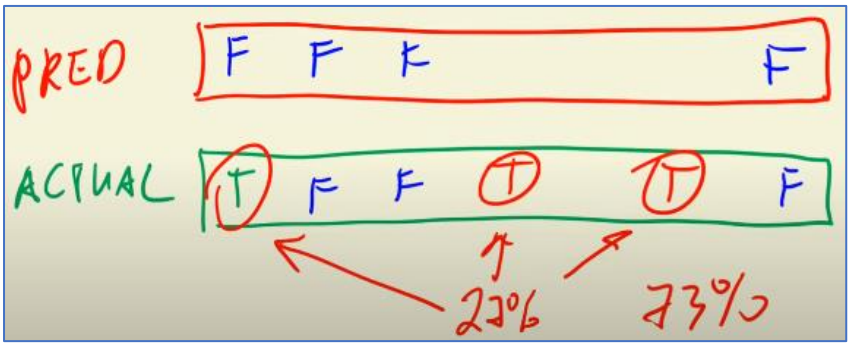
```python
from sklearn.metrics import accuracy_score

thresholds = np.linspace(0, 1, 21)
scores = []

for t in thresholds:
    score = accuracy_score(y_val, y_pred >= t)
    print('%.2f %.3f' % (t, score))
    scores.append(score)
```

```
# Output:
# 0.00 0.274
# 0.05 0.509
# 0.10 0.591
# 0.15 0.666
# 0.20 0.710
# 0.25 0.739
# 0.30 0.760
# 0.35 0.772
# 0.40 0.785
# 0.45 0.793
# 0.50 0.803
# 0.55 0.801
# 0.60 0.795
# 0.65 0.786
# 0.70 0.766
# 0.75 0.744
# 0.80 0.735
# 0.85 0.726
# 0.90 0.726
# 0.95 0.726
# 1.00 0.726
```

Dummy Baseline & Its Accuracy

Observe from the accuracy thresholds table, we get accuracy of 72.6% for threshold 1. Now, threshold 1 is the Dummy Baseline and since the probability scores of all users will be < 1 always, a model with such threshold predicts that no user will churn and it has a accuracy of 72.6% versus 80.3% for 0.5 threshold which is just a difference of 7%. Then, why bother about sending promotional email to users when our model is 72.6% correct that users will not churn ?

This demonstrates the limitations of accuracy as an evaluation metric. While we may achieve a certain level of accuracy, it doesn't always provide the full picture of a model's performance, especially in cases with imbalanced datasets or when specific types of errors are more critical than others.




```python
from collections import Counter

Counter(y_pred >= 1.0)
# Output: Counter({False: 1409})

# Distribution of y_val
Counter(y_val)

# Output: Counter({0: 1023, 1: 386})

1023 / 1409
# Output: 0.7260468417317246
```

From the image above, since threshold is 1, the Churn Prediction is False for all rows. And, the actual Churn value in Validation set is also False in 73% of the rows. If you see the Validation Set has 1023 False values and only 386 True values which is 1023/1409 = 73%. That means, the Validation dataset is IMBALANCED and the issue in here is called **CLASS IMBALANCE** where the data is not balanced and so gives a false impression and accuracy

In cases of **class imbalance**, the traditional accuracy metric can be misleading. For example, a dummy model that predicts the majority class for all samples can achieve a high accuracy simply by getting most of the samples right for the majority class. However, it will perform poorly in identifying the minority class (in this case, the churning customers), which is often more crucial to predict accurately.

To effectively address class imbalance and evaluate our model, we should consider alternative metrics such as:

1. **Precision**: This metric measures the proportion of true positive predictions among all positive predictions. It is particularly useful when the cost of false positives is high.

2. **Recall**: Recall measures the proportion of true positive predictions among all actual positive instances. It is valuable when the cost of false negatives is significant.
3. **F1-Score**: The F1-Score is the harmonic mean of precision and recall, providing a balanced measure that considers both false positives and false negatives.
4. **Area Under the Receiver Operating Characteristic Curve (AUC-ROC)**: The ROC curve plots the true positive rate against the false positive rate at various threshold settings. The AUC-ROC score assesses the classifier's ability to distinguish between the positive and negative classes, making it particularly useful for imbalanced datasets.

Selecting the appropriate evaluation metric depends on the specific goals and requirements of the problem. In cases of class imbalance, accurate identification of the minority class (churning customers) is crucial.

Cases where the threshold may not be kept 0.5

Though thresholds of 0.5 is by default a good choice in most usecases, in medical science like detecting cancer, or any other deadly usecase, it would be good to set the threshold to a lower value like 0.25 or 0.30 because we would not want to classify probability 0.3 of having cancer as "No Cancer" and would want to treat the patient because the cancer probability is still 30% and we can address it sooner

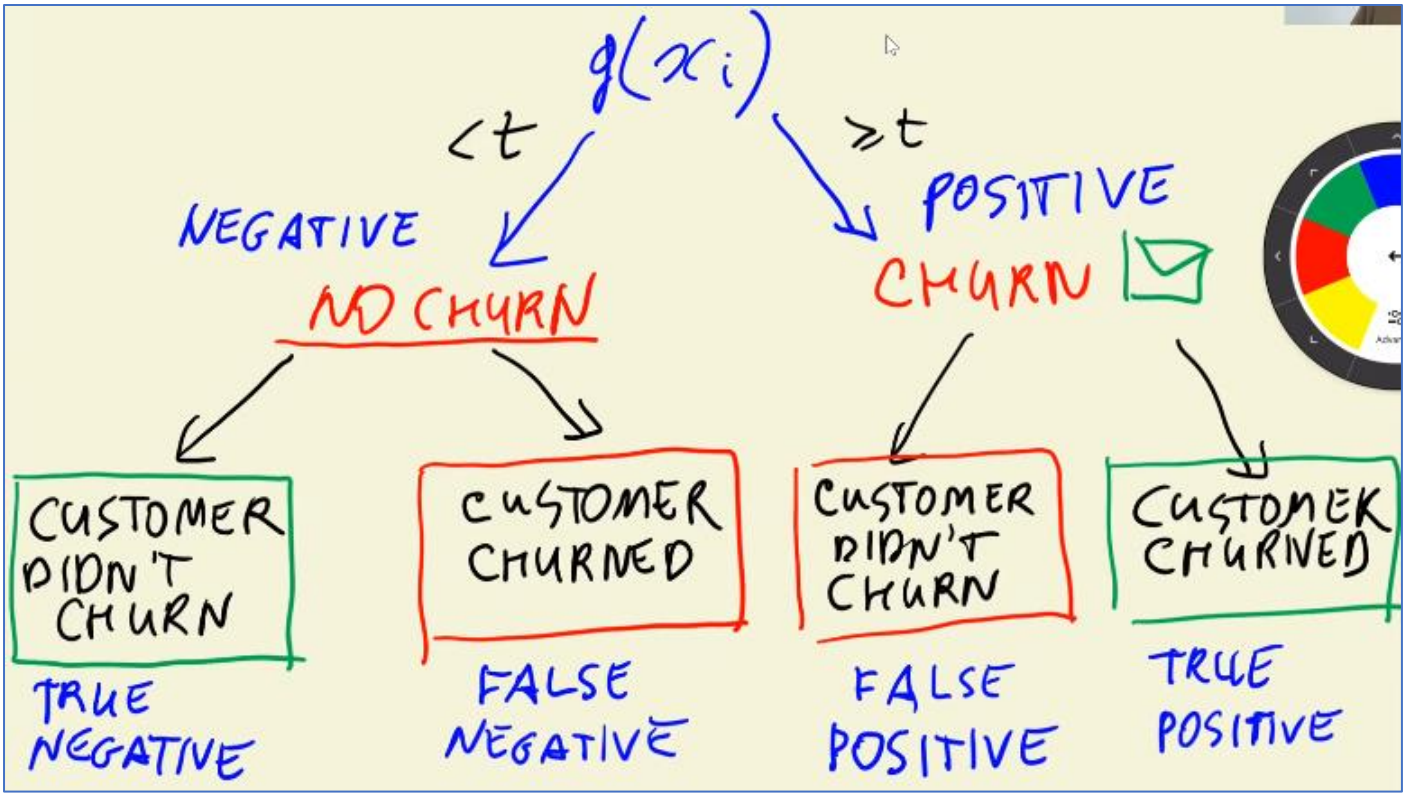
CONFUSION TABLE/MATRIX

Concept

Confusion matrix is a vital tool for evaluating the performance of binary classification models. It allows us to examine the various errors and correct decisions made by our model. As we've previously discussed, class imbalance can significantly impact the accuracy metric. To address this issue, we need alternative evaluation methods that provide a more comprehensive view of our model's performance.

POSITIVE CLASS: CHURN, NEGATIVE CLASS: NO CHURN

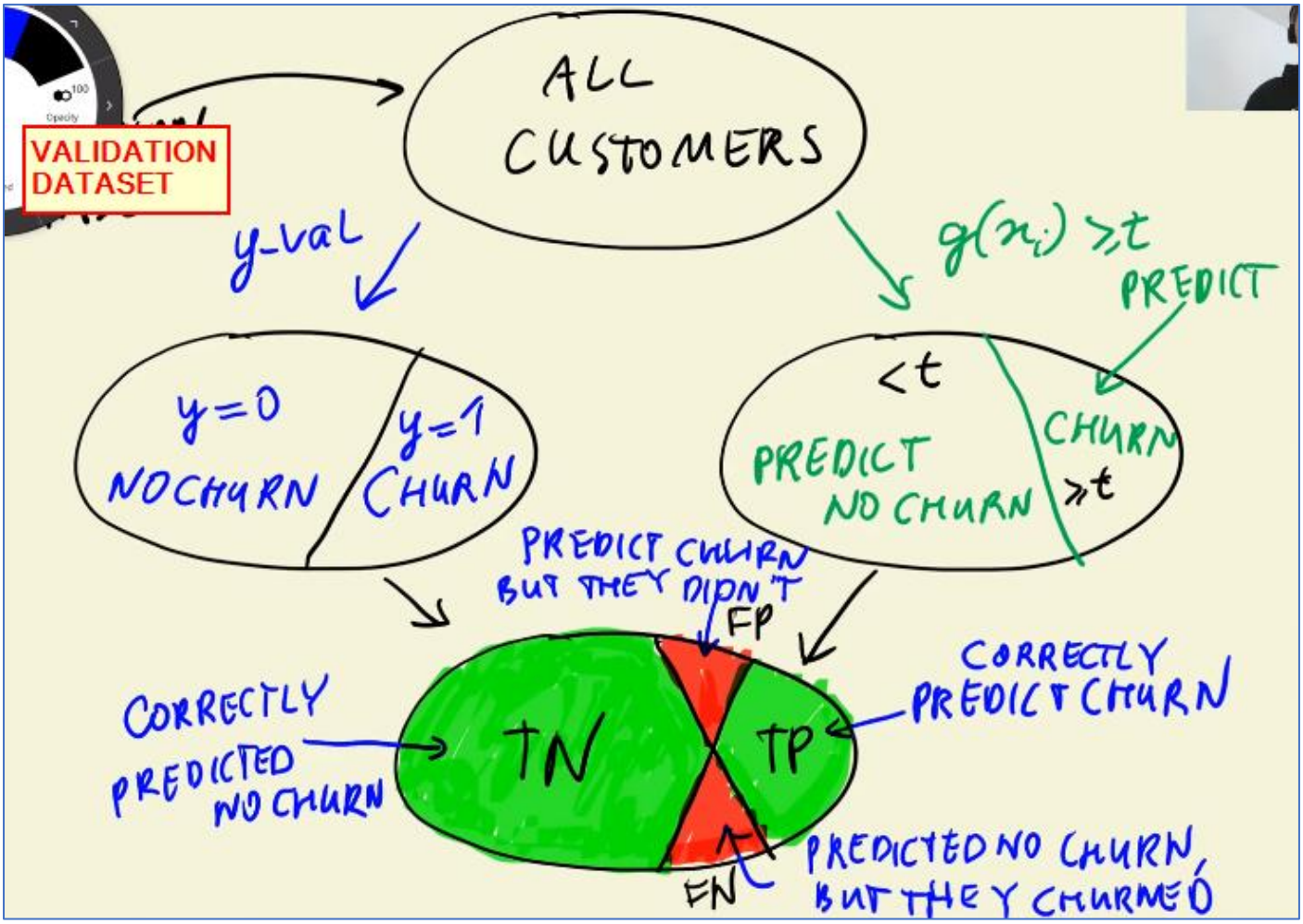
The confusion matrix breaks down the model's predictions into four categories:

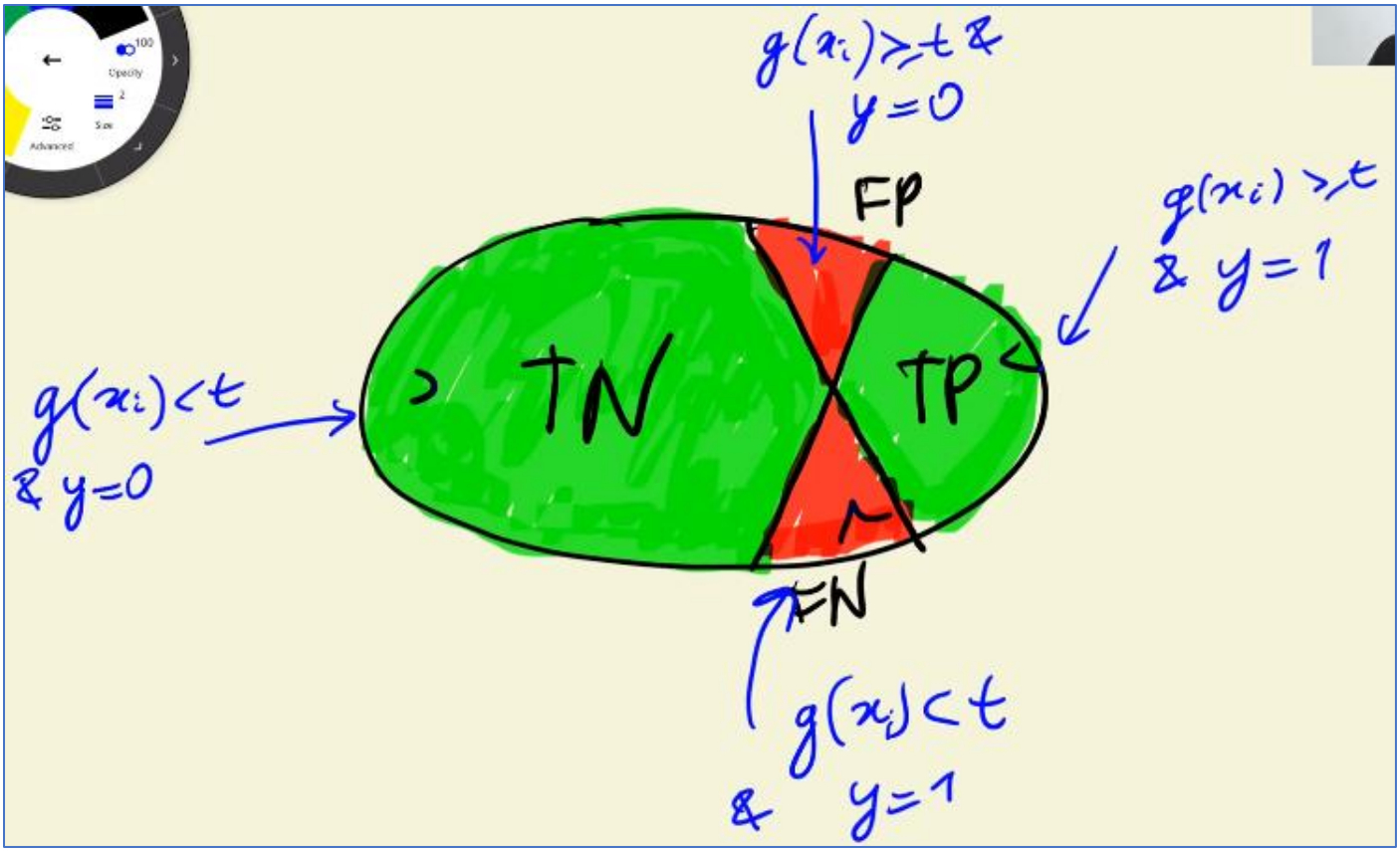


1. **True Positives (TP)**: These are cases where the model correctly predicted the positive class (churning customers) i.e. y_pred > t, y_val =1. Customer was predicted to churn and he did churn

2. **True Negatives (TN)**: These are cases where the model correctly predicted the negative class (non-churning customers) i.e. y_pred < t, y_val =0. Customer was predicted to not churn and he did not churn
3. **False Positives (FP)**: These are cases where the model incorrectly predicted the positive class when the true class was negative i.e. y_pred > t, y_val =0. This is also known as a Type I error. Customer was predicted to churn but he did not churn
4. **False Negatives (FN)**: These are cases where the model incorrectly predicted the negative class when the true class was positive i.e. y_pred < t, y_val =1. This is also known as a Type II error. Customer was predicted to not churn but he churned

We can visualize it through Venn Diagram where we have all the customers. We combine the sectional break-up of y-val, y-pred into one. The non-overlapping parts are correctly predicted as TRUE Predictions. The overlapping parts are incorrect predictions as FALSE Predictions





The Venn Diagram above shows the breakup of all the customers.

True Positives(TP) and True Negatives (TN) are the correct decisions while False Positives (FP) and False Negatives (FN) are incorrect decisions

We can get the numbers for these categories using below Numpy array code-

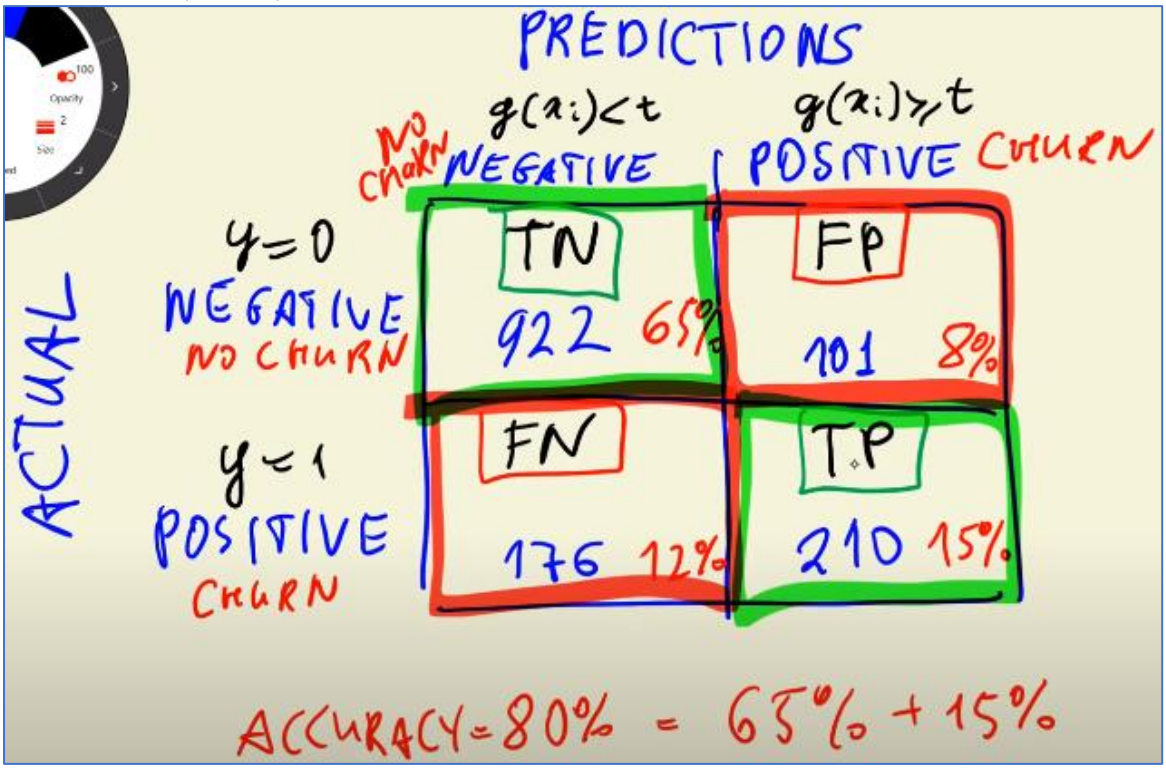
```
# people who are going to churn
actual_positive = (y_val == 1)
# people who are not going to churn
actual_negative = (y_val == 0)
```

```
t = 0.5
predict_positive = (y_pred >= t)
predict_negative = (y_pred < t)
```

And then you can get the numbers for TP, TN, FP, FN using logical & operator and then just summing them (basically will count the number of 1s or True in each Numpy array)

```
predict_positive & actual_positive
# Output: array([False, False, False, ..., False,  True,  True])

tp = (predict_positive & actual_positive).sum()
tp
# Output: 210
tn = (predict_negative & actual_negative).sum()
tn
# Output: 922

fp = (predict_positive & actual_negative).sum()
fp
# Output: 101
fn = (predict_negative & actual_positive).sum()
fn
# Output: 176
```

Tabular Form (Matrix)



We put these 4 numbers (TP, TN, FP, FN) in a 2x2 matrix.

- In the columns of this table, we have the predictions (NEGATIVE: g(xi) < t and POSITIVE: g(xi) >= t).
- In the rows, we have the actual values (NEGATIVE: y=0 and POSITIVE: y=1).
- Way to remember -
  Begin with 0 in both right and down directions (0-Negative, 1-Positive) like in Boolean table. Column is for Prediction, Row is for Actual. First, fill TX where both Pred=Actual i.e. TP, TN. Then, fill FX logically where Pred<>Actual

		PREDICTION	
		0	1
ACTUAL	0	TN	FP
	1	FN	TP

```
confusion_matrix = np.array([
    [tn, fp],
    [fn, tp]
])
```

```
confusion_matrix
# Output:
# array([[922, 101],
#        [176, 210]])
```

```
(confusion_matrix / confusion_matrix.sum()).round(2)
# Output:
# array([[0.65, 0.07],
#        [0.12, 0.15]])
```

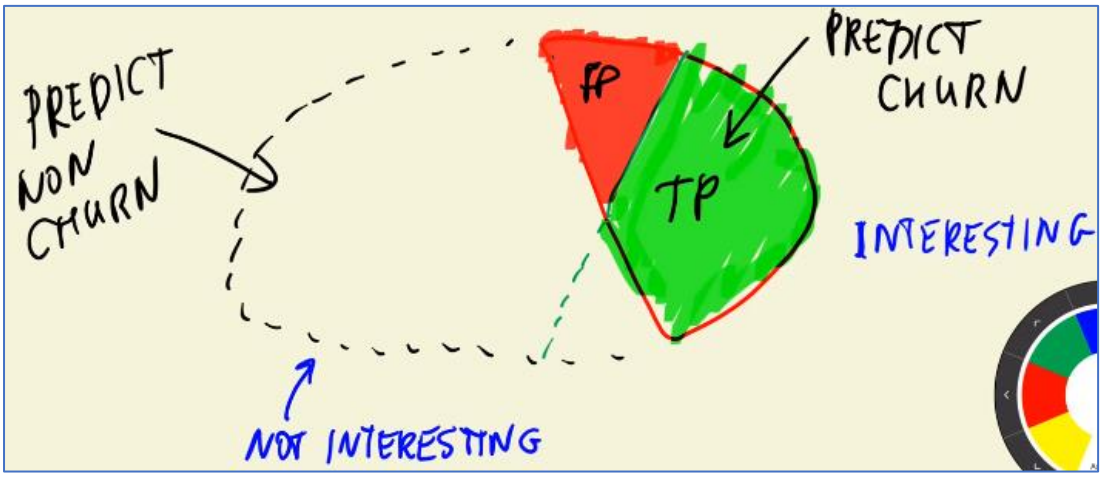
We observe that we have more false negatives than false positives. False positives represent customers who receive the email even though they are not likely to churn, resulting in a loss of money due to unnecessary discounts. False negatives are customers who do not receive the email and end up leaving, causing financial losses as well. Both situations are undesirable. Using the Confusion Matrix numbers, we derive the other metrics like Precision and Recall, Row Curves

PRECISION & RECALL

Precision measures the **fraction of positive predictions that were correct**. In other words, it quantifies how accurately the model predicts customers who are likely to churn. It means that we predict some

customers as churning and then out of those how many are identified correctly

Precision = True Positives / (# Positive Predictions) = True Positives / (True Positives + False Positives) = TP / (TP+FP)




```
## Accuracy = All True / All predictions
accuracy = (tp + tn) / (tp + tn + fp + fn)
accuracy
```

✓ 0.0s

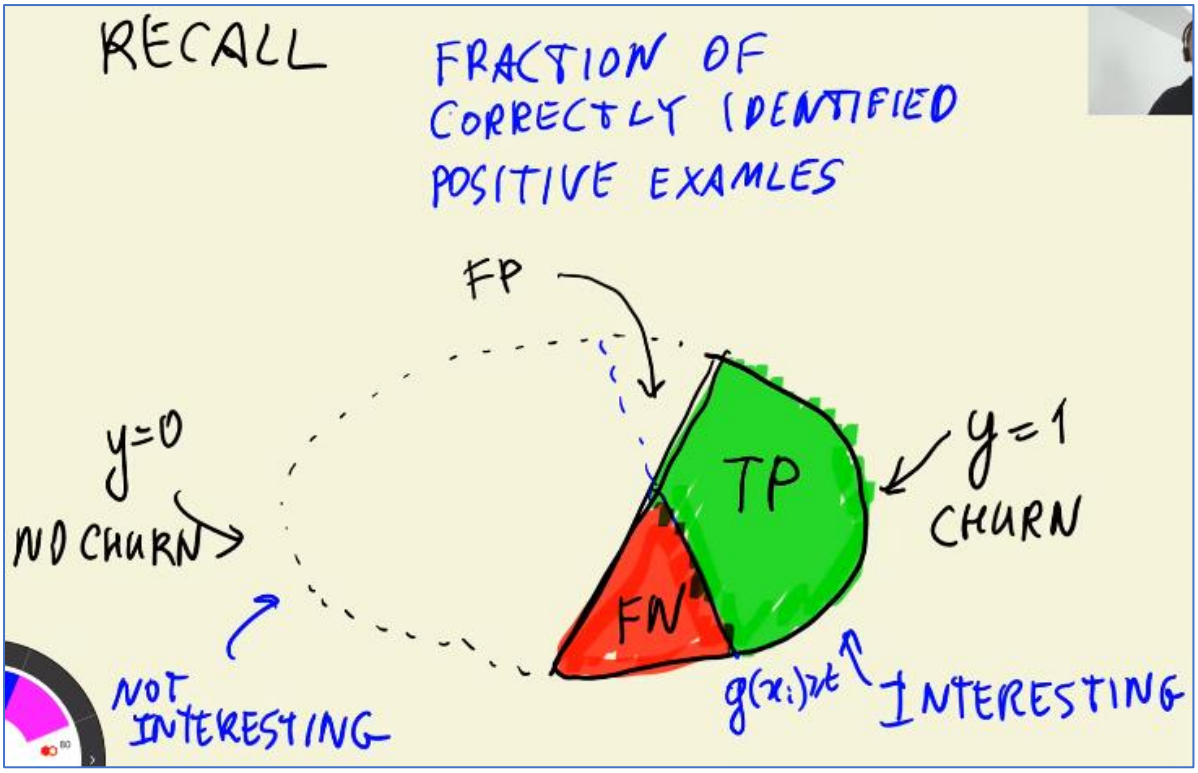
```
0.8034066713981547
```

```
### Precision = True Positive / Positive Predictions
p = tp / (tp + fp)
p
```

```
0.6752411575562701
```

When you say Precision is 67%, it means that out of all users who were predicted to churn, only 67% of those actually churned → there is 33 % mistake

Recall, on the other hand, quantifies the **fraction of actual positive cases that were correctly identified** by the model. It assesses how effectively the model captures all customers who are actually churning.



Recall = True Positives / (# Positive Actuals{Observations}) = True Positives / (True Positives + False Negatives) = TP / (TP + FN)

```
### Recall = True Positive/ Positive Actuals(observations)
r = tp / (tp + fn)
r
### 0.544
```

When you say Recall is 54%, it means that out of users who actually churned, the model predicted only 54% out of them correctly

In summary, precision focuses on the accuracy of positive predictions, while recall emphasizes the model's ability to capture all positive cases. These metrics are crucial for understanding the trade-offs between correctly identifying churning customers and minimizing false positives.

When you have CLASS IMBALANCE, accuracy can give a misleading impression of a model's performance. In such cases, metrics like precision and recall provide a more detailed understanding of how well the model is performing in identifying positive cases (in this case, churning customers).

In scenarios where correctly identifying specific cases is critical, such as identifying churning customers to prevent loss, precision and recall help us make more informed decisions and assess the trade-offs between correctly identifying positives and minimizing false positives or false negatives. So, relying solely on accuracy may not provide a complete picture of a model's effectiveness for a particular task.

MNEMONICS:

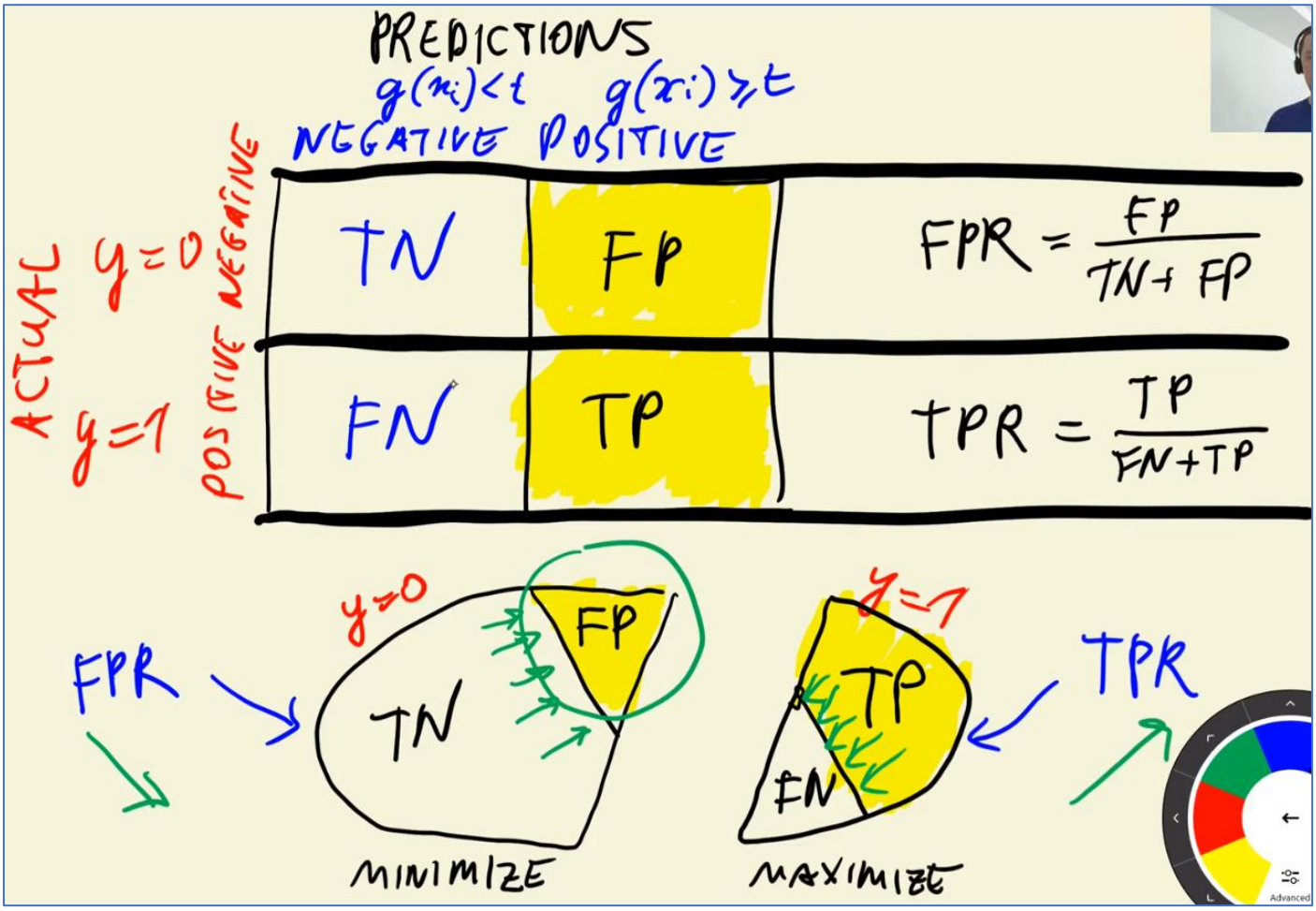
- Precision : From the `pre` dicted positives, how many we predicted right. See how the word `pre` cision is similar to the word `pre` diction?
- Recall : From the `real` positives, how many we predicted right. See how the word `re` c `al` l is similar to the word `real` ?

ROC Curves

ROC (Receiver Operating Characteristic) curves used for evaluating binary classification models, especially in scenarios where you want to assess the trade-off between false positives and true positives at different decision thresholds.

It visually represents the performance of a model by plotting the TPR (True Positive Rate or Sensitivity) against the FPR (False Positive Rate or 1 – Specificity) at various threshold settings. The area under the ROC curve (AUC-ROC) is a summary measure of a model's overall performance, with a higher AUC indicating better discrimination between positive and negative cases.

Concept of ROC Curves



Above figure shows the Confusion matrix at the top from which we derive the calculation for FPR & TPR. Bottom part represents FPR, TPR in the Venn diagram

False positive rate is the fraction of false positives among all actual negative observations. So, we are interested in the first row of the Confusion matrix as it has all the actual negative observations.

$$\mathbf{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

And, we want to minimize the FPR as much as possible, so reduce FP

True Positive Rate is the fraction of True Positives amongst all actual positive observations. So, we are interested in the second row of the Confusion matrix as it has all the actual positive observations.
It is also called SENSITIVITY or RECALL

$$Sensitivity/TPR = \frac{TP}{TP + FN}$$

And, we want to maximize the TPR as much as possible, so increase TP

Calculation of ROC Curves

First, we calculate the FPR, TPR as per the formula. Now, this is for threshold = 0.5

```
tpr = tp / (tp + fn)
tpr
# Output: 0.5440414507772021
recall
# Output: 0.5440414507772021
# --> tpr = recall

fpr = fp / (fp + tn)
fpr
# Output: 0.09872922776148582
```

Now, we extend this formula to compute FPR, TPR for different thresholds. The more the number of thresholds we compute, the better granular graph we would get.

So, we do it for 100 thresholds between 0 to 1 using np.linspace(0,1,101) each threshold incrementing by 0.01. First, we compute the numbers TP, FP, FN, TN and add them as tuple to "scores" list variable

```
scores = []
thresholds = np.linspace(0, 1, 101)

for t in thresholds:
    actual_positive = (y_val == 1)
    actual_negative = (y_val == 0)

    predict_positive = (y_pred >= t)
    predict_negative = (y_pred < t)

    tp = (predict_positive & actual_positive).sum()
    tn = (predict_negative & actual_negative).sum()

    fp = (predict_positive & actual_negative).sum()
    fn = (predict_negative & actual_positive).sum()

    scores.append((t, tp, tn, fp, fn))

scores
```

```
# Output:
# [(0.0, 386, 0, 1023, 0),
# (0.01, 385, 110, 913, 1),
# (0.02, 384, 193, 830, 2),
# (0.03, 383, 257, 766, 3),
# (0.04, 381, 308, 715, 5),
# (0.05, 379, 338, 685, 7),
# (0.06, 377, 362, 661, 9),
# (0.07, 372, 382, 641, 14),
# (0.08, 371, 410, 613, 15),
# (0.09, 369, 443, 580, 17),
# (0.1, 366, 467, 556, 20),
# (0.11, 365, 495, 528, 21),
# (0.12, 365, 514, 509, 21),
# (0.13, 360, 546, 477, 26),
# (0.14, 355, 570, 453, 31),
# (0.15, 351, 588, 435, 35),
# (0.16, 347, 604, 419, 39),
# (0.17, 346, 622, 401, 40),
# (0.18, 344, 639, 384, 42),
# (0.19, 338, 654, 369, 48),
# (0.2, 333, 667, 356, 53),
# (0.21, 330, 682, 341, 56),
# (0.22, 323, 701, 322, 63),
# (0.23, 320, 710, 313, 66),
# (0.24, 316, 719, 304, 70),
# ...
# (0.96, 0, 1023, 0, 386),
# (0.97, 0, 1023, 0, 386),
# (0.98, 0, 1023, 0, 386),
# (0.99, 0, 1023, 0, 386),
# (1.0, 0, 1023, 0, 386)]
```

We end up with 101 confusion matrices evaluated for different thresholds. Let's turn that into a dataframe.

```
columns = ['threshold', 'tp', 'tn', 'fp', 'fn']
df_scores = pd.DataFrame(scores, columns=columns)
df_scores
```

	threshold	TP	TN	FP	FN
0	0.00	386	0	1023	0
1	0.01	385	110	913	1
2	0.02	384	193	830	2
3	0.03	383	257	766	3
4	0.04	381	308	715	5
...	...	...	...	...	...
96	0.96	0	1023	0	386
97	0.97	0	1023	0	386
98	0.98	0	1023	0	386
99	0.99	0	1023	0	386
100	1.00	0	1023	0	386

We can look at each tenth record by using this ::10 operator. This works by printing every record starting from the first record till end, moving forward with increments of 10.

```
df_scores[::10]
```

	threshold	tp	tn	fp	fn
0	0.0	386	0	1023	0
10	0.1	366	467	556	20
20	0.2	333	667	356	53
30	0.3	284	787	236	102
40	0.4	249	857	166	137
50	0.5	210	922	101	176
60	0.6	150	970	53	236
70	0.7	76	1003	20	310
80	0.8	13	1022	1	373
90	0.9	0	1023	0	386
100	1.0	0	1023	0	386

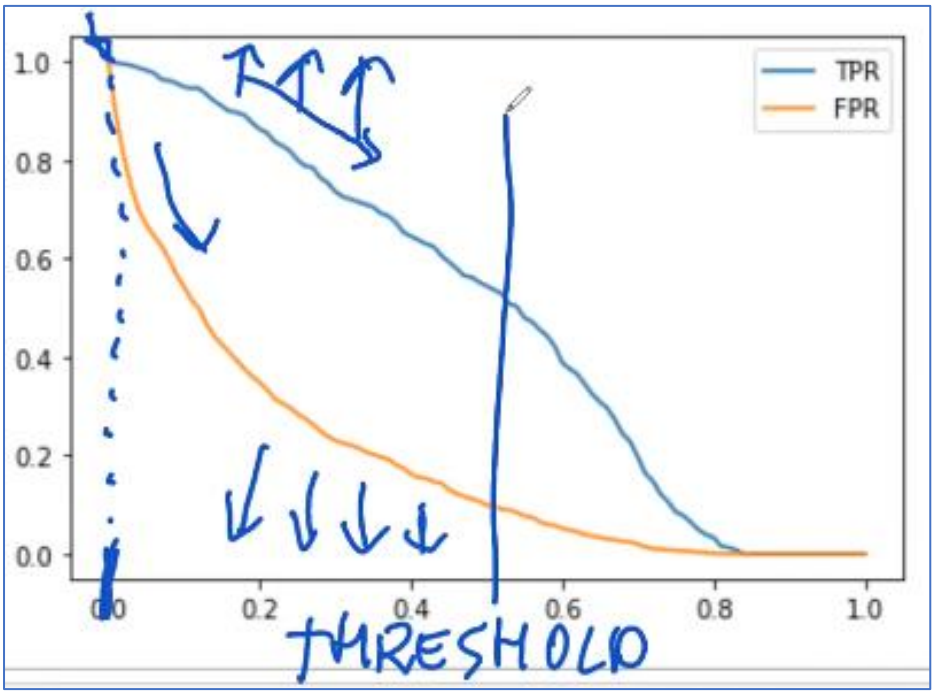
Add new columns “tpr”, “fpr” to the dataframe

```
df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)
df_scores[::10]
```

	threshold	tp	TN	FP	FN	tpr	fpr
0	0.0	386	0	1023	0	1.000000	1.000000
10	0.1	366	467	556	20	0.948187	0.543500
20	0.2	333	667	356	53	0.862694	0.347996
30	0.3	284	787	236	102	0.735751	0.230694
40	0.4	249	857	166	137	0.645078	0.162268
50	0.5	210	922	101	176	0.544041	0.098729
60	0.6	150	970	53	236	0.388601	0.051808
70	0.7	76	1003	20	310	0.196891	0.019550
80	0.8	13	1022	1	373	0.033679	0.000978
90	0.9	0	1023	0	386	0.000000	0.000000
100	1.0	0	1023	0	386	0.000000	0.000000

Then, we plot the TPR, FPR on y-axis, and thresholds on x-axis

```
plt.plot(df_scores.threshold, df_scores['tpr'], label='TPR')
plt.plot(df_scores.threshold, df_scores['fpr'], label='FPR')
plt.legend()
```



Graph interpretation - For threshold of 0.0, all predictions are POSITIVE (all churning) and all the POSITIVE ACTUALS will be correctly identified (no false negative). So, TPR = TP/(TP+0) = 1. Also, out of all the NEGATIVE ACTUALS, all of them will be FALSE POSTIVES (no true negative) and so FPR = FP/(FP+0) = 1
As threshold increases, FPR decreases quickly compared to TPR and that is what we want that FPR should be minimum and ideally, we want to keep TPR as high (closer to 1) as possible. For threshold of 0.5, we see a TPR around 50% and FPR around 10%. To determine how good our model performs, we need to compare this ROC Curve with that for RANDOM Model & IDEAL MODEL

Comparison with RANDOM MODEL

Random model is one where we flip a coin to determine POSITIVE/NEGATIVE outcome. We use a UNIFORM Random Distribution to generate new Validation Set (y_rand) of same size as Original Validation set (y_val)

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
y_rand.round(3)
# Output: array([0.417, 0.72 , 0.   , ..., 0.774, 0.334, 0.089])
```

```
# Accuracy for our random model is around 50%
((y_rand >= 0.5) == y_val).mean()

# Output: 0.5017743080198722
```

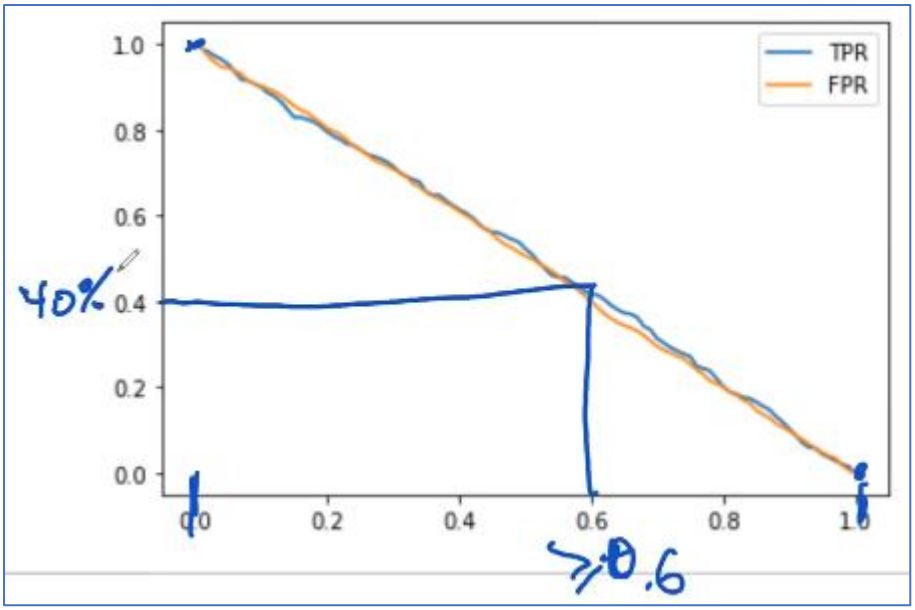
Lets put the previous code of TPR, FPR calculations into a function tpr_fpr_dataframe(y_val, y_pred). And then have TPR, FPR dataframe df_rand for the Random model by passing y_val, y_rand. And then plot the ROC graph

```
def tpr_fpr_dataframe(y_val, y_pred):
    scores = []
    thresholds = np.linspace(0, 1, 101)

    for t in thresholds:
        actual_positive = (y_val == 1)
        actual_negative = (y_val == 0)

        predict_positive = (y_pred >= t)
        predict_negative = (y_pred < t)

        tp = (predict_positive & actual_positive).sum()
        tn = (predict_negative & actual_negative).sum()

        fp = (predict_positive & actual_negative).sum()
        fn = (predict_negative & actual_positive).sum()

        scores.append((t, tp, tn, fp, fn))

    columns = ['threshold', 'tp', 'tn', 'fp', 'fn']
    df_scores = pd.DataFrame(scores, columns=columns)

    df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
    df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)

    return df_scores
```

```
df_rand = tpr_fpr_dataframe(y_val, y_rand)
df_rand[::10]
```

	threshold	tp	tn	fp	fn	tpr	fpr
0	0.0	386	0	1023	0	1.000000	1.000000
10	0.1	347	100	923	39	0.898964	0.902248
20	0.2	307	201	822	79	0.795337	0.803519
30	0.3	276	299	724	110	0.715026	0.707722
40	0.4	237	399	624	149	0.613990	0.609971
50	0.5	202	505	518	184	0.523316	0.506354
60	0.6	161	614	409	225	0.417098	0.399804
70	0.7	121	721	302	265	0.313472	0.295210
80	0.8	78	817	206	308	0.202073	0.201369
90	0.9	40	922	101	346	0.103627	0.098729
100	1.0	0	1023	0	386	0.000000	0.000000

```
plt.plot(df_rand.threshold, df_rand['tpr'], label='TPR')
plt.plot(df_rand.threshold, df_rand['fpr'], label='FPR')
plt.legend()
```

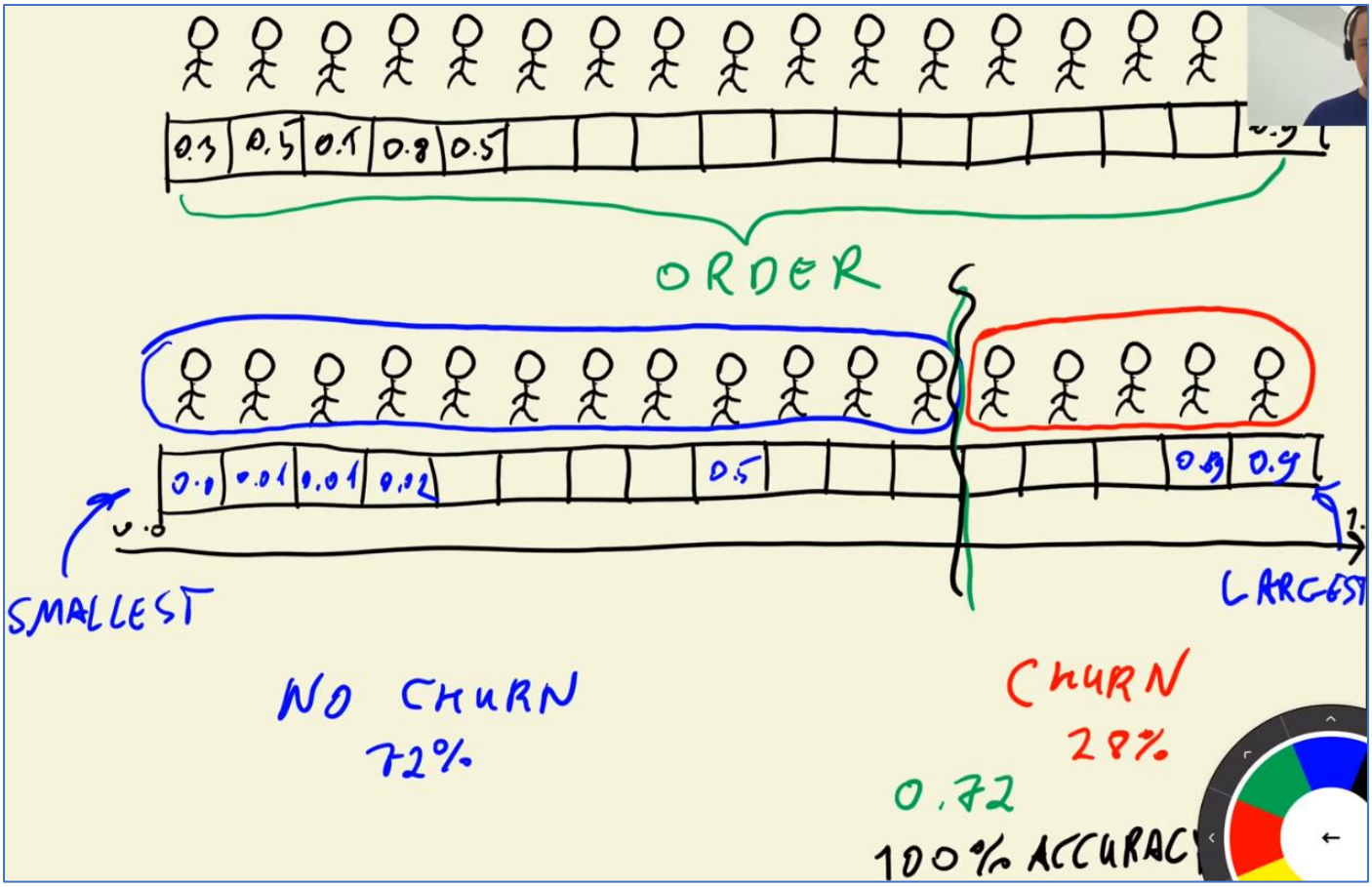


Graph interpretation-

Both TPR, FPR follow a decreasing straight line. For example, at threshold of 0.6, both the TPR, FPR around 40%. The reason behind these values is that our model's predictions are nearly equivalent to tossing a coin. In 60% of cases, the model predicts that a customer is non-churning, and in 40% of cases, it predicts that the customer is churning. In other words, these rates indicate that the model predicts a customer as churning with a 40% probability and as non-churning with a 60% probability. Consequently, the model is incorrect for non-churning customers in 40% of cases.

Comparison with IDEAL MODEL

IDEAL model is one which gets all predictions Correct.



For this, assume we have users as shown in 1st row with probabilities 0.3, 0.5, 0.8, 0.5 ….0.9. We order them in increasing order of probability from 0.0 to 0.9. Ideal model orders the users such that all the non-churning ones are to the left and churning ones are to RIGHT. And probability increases from 0 to 1. If our threshold is 0.72, which is

exactly where the non-churning and churning users are divided, then our model will have 100% accuracy because it will correctly classify non-churning users as non-churning and churning ones as churning

To implement such a model, first need to know the number of negative examples (number of people who are not churning) by counting the number of 0 in y_val and the number of positive examples. First create a y_ideal array that contains only negative observations (0s) followed by positive observations (1s). We use the np.repeat() function to achieve this, creating an array with 1023 zeros and then 386 ones.

```
num_neg = (y_val == 0).sum()
num_pos = (y_val == 1).sum()
num_neg, num_pos

# Output: (1023, 386)
```

```
y_ideal = np.repeat([0, 1], [num_neg, num_pos])
y_ideal

# Output: array([0, 0, 0, ..., 1, 1, 1])
```

To create our predictions for the ideal model, which are numbers between 0 and 1, we can use the np.linspace() function to generate an array of evenly spaced values between 0 and 1. This array should have the same length as y_ideal, which is 1409 in this case.

```
y_ideal_pred = np.linspace(0, 1, len(y_ideal))
y_ideal_pred

# Output:
# array([0.00000000e+00, 7.10227273e-04, 1.42045455e-03, ...,
#        9.98579545e-01, 9.99289773e-01, 1.00000000e+00])
```

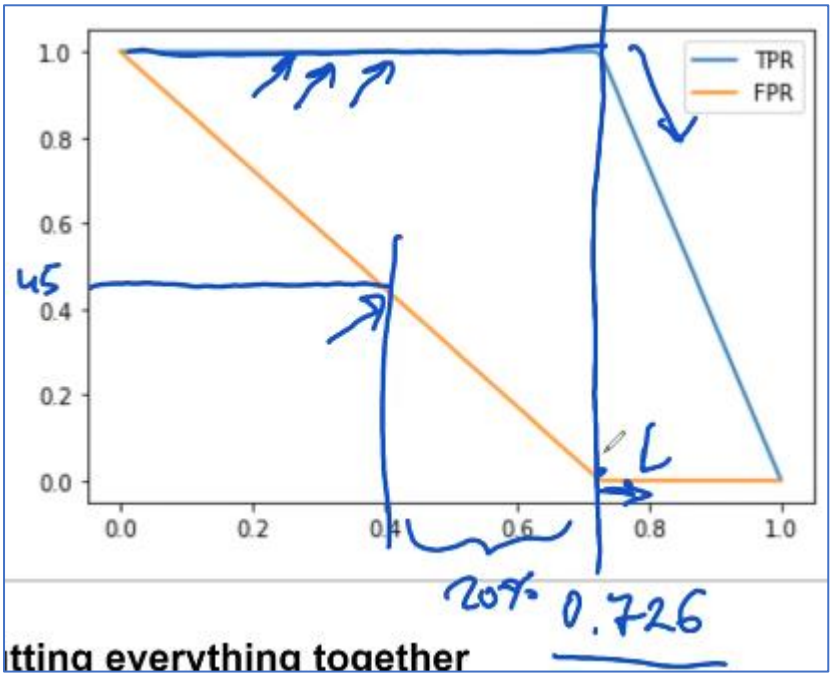
```
1 - y_val.mean()
# Output: 0.7260468417317246



accuracy_ideal = ((y_ideal_pred >= 0.726) == y_ideal).mean()
accuracy_ideal

# Output: 1.0
```

With threshold set to 0.726, this model has 100% accuracy and is termed the ideal model.
The ideal model, which makes perfect predictions, doesn't exist in reality, but it serves as a benchmark to understand how well our actual model is performing. By comparing our model's performance to that of the ideal model, we can assess how much room for improvement there is.
We do the same exercise for the ideal model as we did for the random model so we'll create a data frame df_ideal

```
df_ideal = tpr_fpr_dataframe(y_ideal, y_ideal_pred)
df_ideal[::10]
```

	threshold	tp	tn	fp	fn	tpr	fpr
0	0.0	386	0	1023	0	1.000000	1.000000
10	0.1	386	141	882	0	1.000000	0.862170
20	0.2	386	282	741	0	1.000000	0.724340
30	0.3	386	423	600	0	1.000000	0.586510
40	0.4	386	564	459	0	1.000000	0.448680
50	0.5	386	704	319	0	1.000000	0.311828
60	0.6	386	845	178	0	1.000000	0.173998
70	0.7	386	986	37	0	1.000000	0.036168
80	0.8	282	1023	0	104	0.730570	0.000000
90	0.9	141	1023	0	245	0.365285	0.000000
100	1.0	1	1023	0	385	0.002591	0.000000

```
plt.plot(df_ideal.threshold, df_ideal['tpr'], label='TPR')
plt.plot(df_ideal.threshold, df_ideal['fpr'], label='FPR')
plt.legend()
```



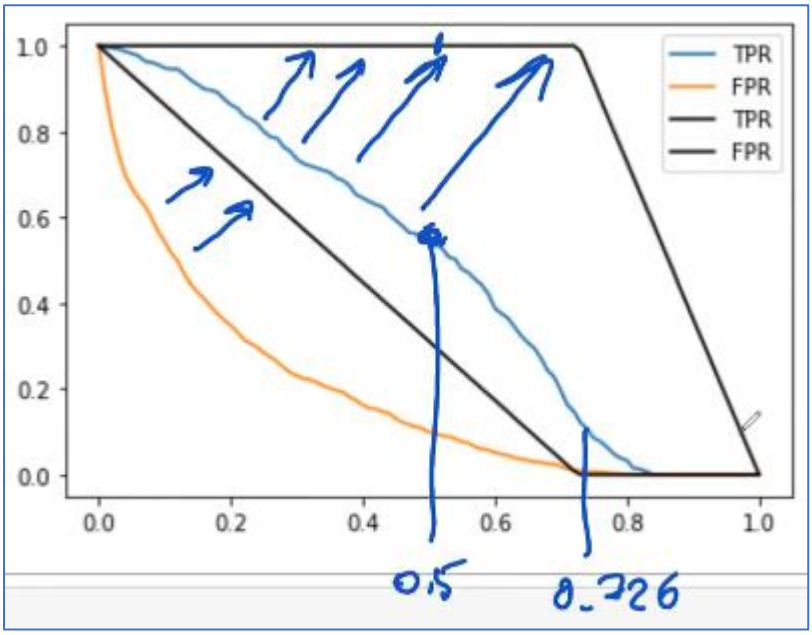
Graph interpretation-

TPR stays 1 uptil the threshold value 0.726. At threshold value 0.726, FPR is 0. At threshold of say 0.4, FPR is 45% which means the model is making mistakes

So, now we have two benchmarks for ROC curves - Random model and Ideal model. To know how good our model performance is on the ROC curve, we plot the curves for Random, ideal and our model together

```
plt.plot(df_scores.threshold, df_scores['tpr'], label='TPR')
plt.plot(df_scores.threshold, df_scores['fpr'], label='FPR')

#plt.plot(df_rand.threshold, df_rand['tpr'], label='TPR')
#plt.plot(df_rand.threshold, df_rand['fpr'], label='FPR')

plt.plot(df_ideal.threshold, df_ideal['tpr'], label='TPR', color = 'black')
plt.plot(df_ideal.threshold, df_ideal['fpr'], label='FPR', color = 'black')

plt.legend()
```

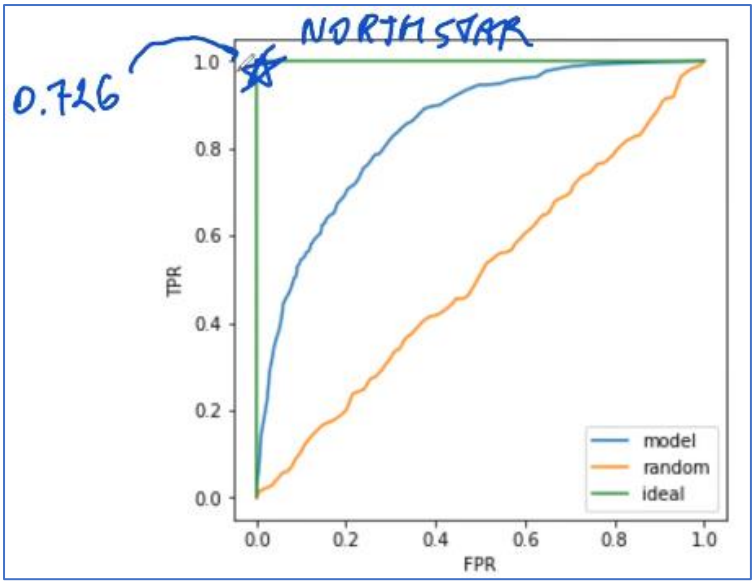
Since we know that the plot for Random model is just a straight line from 1 to 0 as threshold increases from 0 to 1, we ignore it from the plot to focus on the plots for Ideal and our model

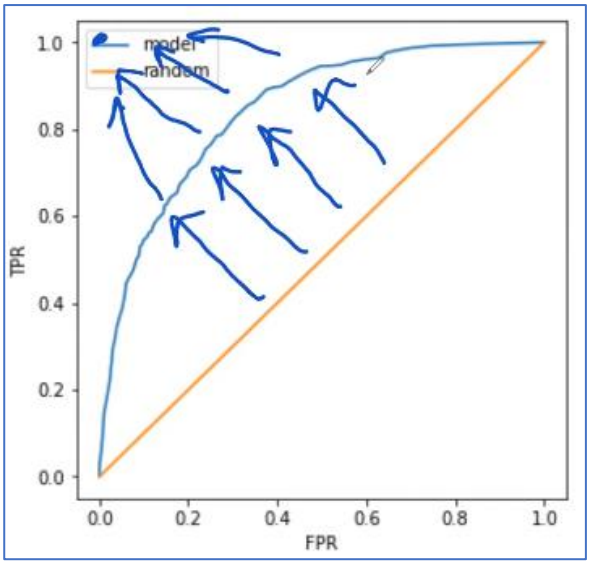


Graph Interpretation- compare our model with the ideal model, our TPR true positive rate we actually want to be as close as possible to the ideal TPR = one right so we see that it's quite far apart at the threshold 0.5 means we are making quite a lot of mistakes and likewise this false positive rate is also quite far from ideal FPR. Plotting against the threshold does not give correct picture. For example, in our model, the best threshold is 0.5, as we know from accuracy. However, for the ideal model, as we saw earlier, the best threshold is 0.726. So they have different thresholds.

What we can do to better visualize this is to plot FPR against TPR. On the x-axis, we'll have FPR, and on the y-axis, we'll have TPR → this is the **ROC Curve**

```
plt.figure(figsize=(5,5))

plt.plot(df_scores.fpr, df_scores.tpr, label='model')
plt.plot([0,1], [0,1], label='random')
#plt.plot(df_rand.fpr, df_rand.tpr, label='random')
#plt.plot(df_ideal.fpr, df_ideal.tpr, label='ideal')

plt.xlabel('FPR')
plt.ylabel('TPR')

plt.legend()
```



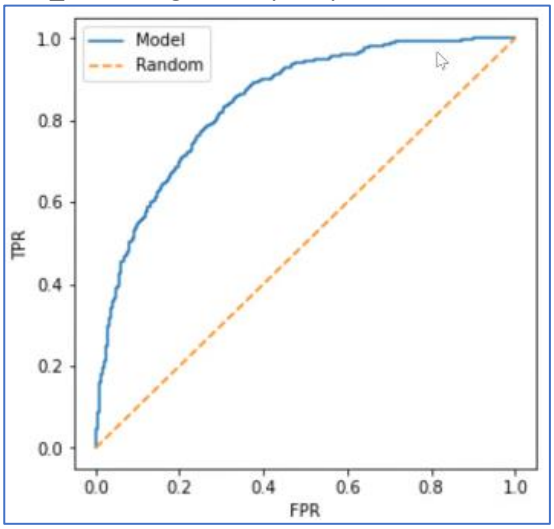


Graph interpretation- For ideal model (green line), where the TPR = 100% and FPR= 0%, that is the ideal point (NOTRH STAR) where we want to get. Since we know how the ideal line looks like, we can remove it from the plot and then see random and our model. So, we want our ROC curve to be closer to the NORTH STAR and far from the Random plot. If our model closely resembles the random baseline model, then it is not performing well.

ROC from scikit learn

Instead of calculating ROC by our own, we can use scikit learn function roc_curve to get the tpr, frp and thresholds

```
# We can also use the ROC functionality of scikit learn package
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_val, y_pred)

plt.figure(figsize=(5,5))

plt.plot(fpr, tpr, label='Model')
plt.plot([0,1], [0,1], label='Random', linestyle='--')

plt.xlabel('FPR')
plt.ylabel('TPR')

plt.legend()
```



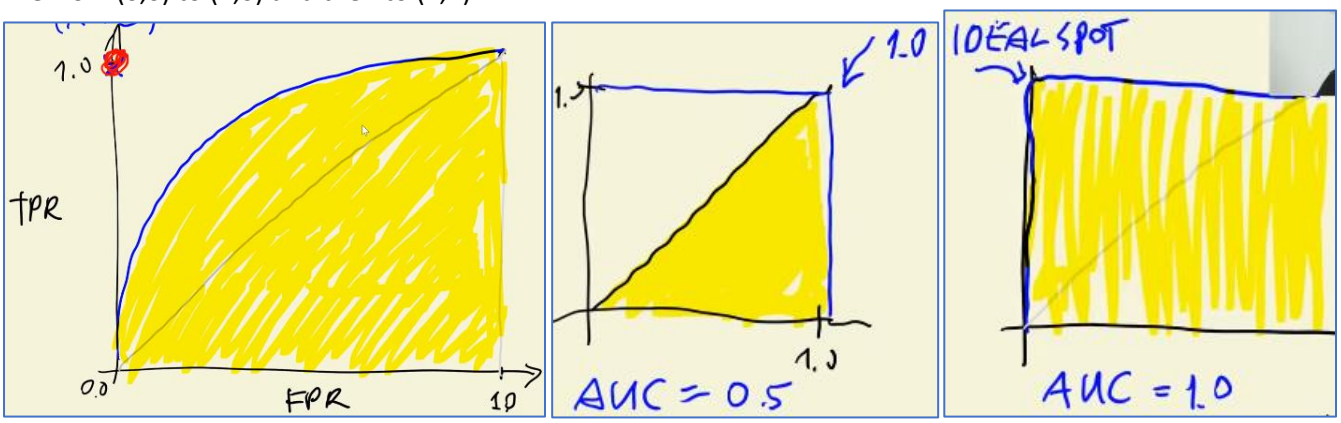
What kind of information do we get from ROC curve?

Let's begin in the lower-left corner, where both TPR and FPR are 0. This occurs at higher thresholds like 1.0. In this scenario, we predict that every customer is non-churning, resulting in TPR being 0 since we don't predict anyone as churning. FPR is also 0 because there are no false positives; we only have true negatives (TN).
As we move from the lower left corner, where the threshold starts at 1.0, we eventually reach the upper-right corner with a threshold of 0.0. Here, our model achieves 100% TPR because we predict everyone as churning, enabling us to identify all churning customers. However, we also make many mistakes, incorrectly identifying non-churning customers. Thus, we have TPR = FPR = 100%.
When we adjust the threshold, we predict more customers as churning, causing our TPR to increase, but the FPR also increases concurrently.

The ROC curve allows us to observe how the model behaves at different thresholds. Each point on the ROC curve represents TPR and FPR evaluated at a specific threshold. By plotting this curve, we can assess how far the model is from the ideal spot and how far it is from the random baseline. Additionally, the ROC curve is useful for comparing different models, as it's easy to determine which one is superior (a model closer to the ideal spot is better, while one closer to the random baseline is worse).

AUC - Area under the Curve for ROC Curves

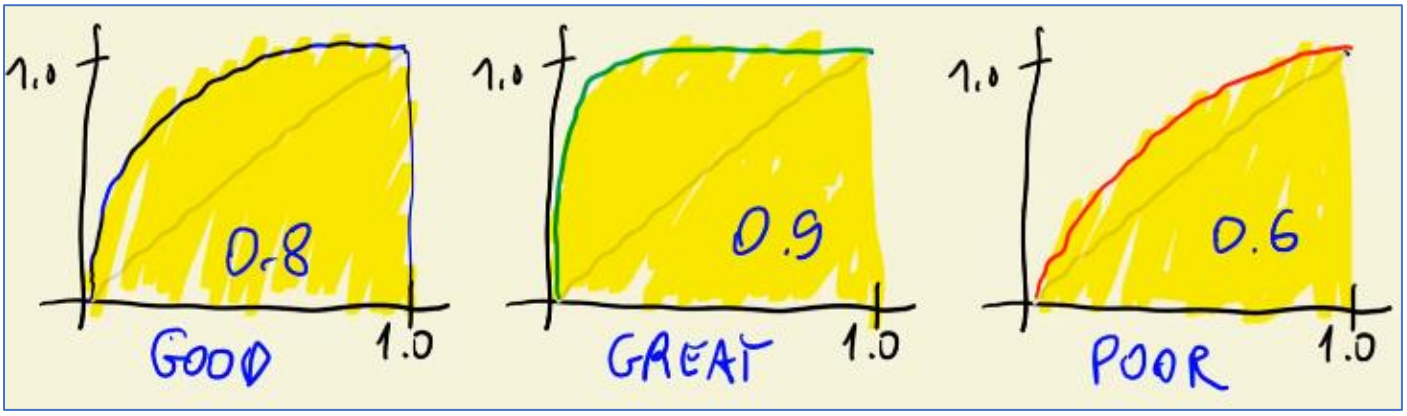
From the concept and interpretation of ROC Curve, we understand that we want the curve of our model to move towards the top-left corner (which is the ideal curve). But, how much close/far is it from the ideal curve can be quantified by metric called "AREA UNDER THE CURVE" (highlighted in yellow).
Random baseline is the straight line from bottom-left corner to top-right corner while the ideal model is the straight line from (0,0) to (1,0) and then to (1,1)



So, AUC for Random baseline = 0.5 (half of 1x1 square) and for Ideal = 1.0

So, **0.5 < AUC for our model < 1.0**

AUC = 0.8 (Good), AUC = 0.9 (GREAT), AUC = 0.6 (POOR), AUC < 0.5 (Mistake)



AUC Calculation

Scikit learn has function to calculate Area under any Curve (and not only ROC Curve). sklearn.metrics.auc()
We pass the fpr, tpr that we got using scikit learn's roc_curve function

```
from sklearn.metrics import auc
# auc needs values for x-axis and y-axis
auc(fpr, tpr)
# Output: 0.843850505725819
```

.

Compare the auc for fpr, tpr as per our manual method with this one, both are very close

```
auc(df_scores.fpr, df_scores.tpr)
# Output: 0.8438732975754537
```

AUC of our manual ideal model is also very close to 1

```
auc(df_ideal.fpr, df_ideal.tpr)
# Output: 0.9999430203759136
```

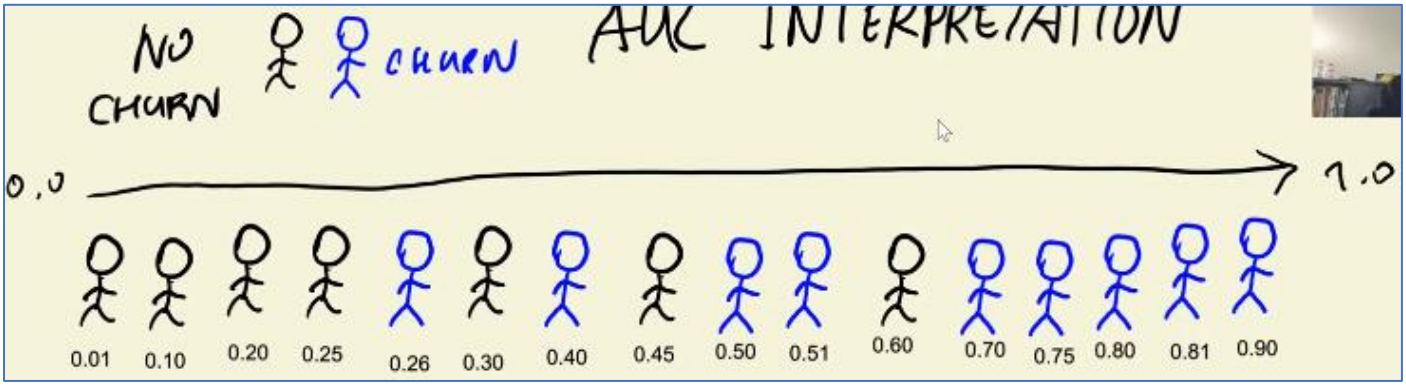
So, from y-val, y-pred, we can directly get the AUC by getting fpr, tpr using roc_curve (y_val, y_pred) function and then auc(fpr, tpr) function OR scikit learn has roc_auc_score(y_val,y_pred) function which do both these steps in one step

```
fpr, tpr, thresholds = roc_curve(y_val, y_pred)
auc(fpr, tpr)

# Output: 0.843850505725819
```

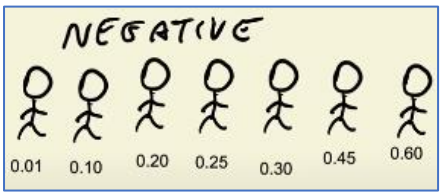
```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_val, y_pred)

# Output: 0.843850505725819
```

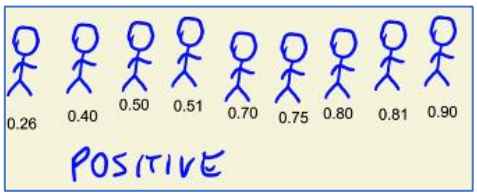
AUC Interpretation

AUC tells us the **Probability that a randomly selected positive example has a Probability score that is higher than a randomly selected negative example**.

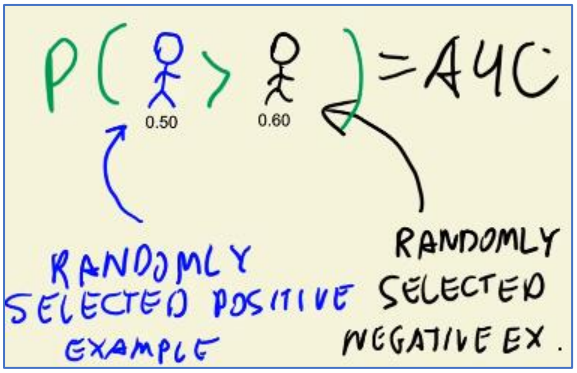


Black ones are actual NO CHURN and Blue ones are actual CHURN. They are ordered by their probability scores of churning. So, we group them into POSITIVE SET and NEGATIVE SET.





And then we randomly select one positive example and one negative example and note if Probability Score of Positive > Probability Score of Negative. We repeat this random selection multiple times and then Probability of getting (P > N) is equal to the AUC



Lets implement this manually-

```
neg = y_pred[y_val == 0]
pos = y_pred[y_val == 1]
```

,

```
import random
pos_ind = random.randint(0, len(pos) -1)
neg_ind = random.randint(0, len(neg) -1)
```

,

```
pos[pos_ind] > neg[neg_ind]
# Output: True
```

neg has all the Probability scores of negative examples; pos has all the Probability scores of positive examples. Then, we use random function to give index between 0 to neg/pos count. Whichever index is selected, we compare the Probability score of pos, neg. We repeat this n multiple times and compute the probability as success/n

```
n = 100000
success = 0

for i in range(n):
    pos_ind = random.randint(0, len(pos) -1)
    neg_ind = random.randint(0, len(neg) -1)

    if pos[pos_ind] > neg[neg_ind]:
        success += 1

success / n

# Output: 0.84389
```

→ This result is quite close to roc_auc_score(y_val, y_pred) = 0.84385

Lets now use Numpy instead. Note that in np.random.randint(low, high, size, dtype), 'low' is inclusive, and 'high' is exclusive.

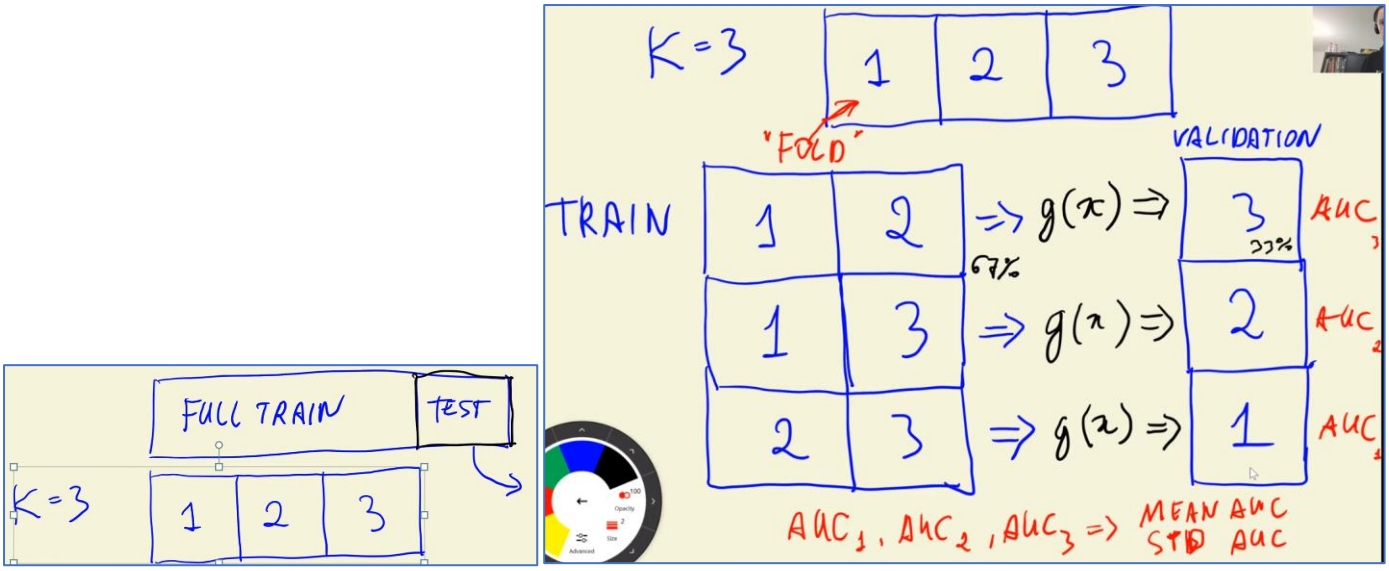
```
n = 50000

np.random.seed(1)
pos_ind = np.random.randint(0, len(pos), size=n)
neg_ind = np.random.randint(0, len(neg), size=n)
pos[pos_ind] > neg[neg_ind]
# Output: array([False,  True,  True, ...,  True,  True,  True])

(pos[pos_ind] > neg[neg_ind]).mean()
# Output: 0.84646
```

pos_ind, neg_ind will be Numpy arrays with each element an index. We then sequentially compare elements in pos, neg using pos_ind, neg_ind. And, to know how many cases are TRUE, we simply take the mean() which gives output 0.846; close to the AUC

So, **<u>AUC is the Performance Metric for BINARY CLASSIFICATION MODELS. Higher the AUC towards 1, better the model</u>**

CROSS-VALIDATION (KFold)

It is the process of training and evaluating a model on different subsets of same dataset. We know that we have to split our dataset into 3 parts - TRAINING, VALIDATION & TEST. We keep Aside the TEST portion and use the VALIDATION Part to measure performance like AUC. → This approach is called **<u>HOLD OUT STRATEGY</u>** (because the Test part is "held out"). TRAIN + VALIDATION ➔ FULL TRAIN. We use the FULL TRAIN to generate the model function g(x) and validate it using VALIDATION Part and we get the metrics



Now, we can take another approach which is called the **K-Fold Cross Validation**. We split the FULL TRAIN into 'k' parts where each part is called a FOLD. Lets take k = 3, so FULL TRAIN is split into 3 Folds. Now, we use Fold 1 & 2 combined to train the model (i.e. get g1(x)) and Fold 3 to validate that model and compute its AUC. Next, we use Fold 1 & 3 combined to train the model (i.e. get g2(x)) and Fold 2 to validate that model and compute its AUC. And then, we use Fold 2 & 3 combined to train the model (i.e. get g3(x)) and Fold 1 to validate that model and compute its AUC.
This way, we get 3 AUCs, we compute their average AUC and its standard deviation. Standard deviation will show how much the AUC differs across different subsets and stability of the model

Implementation

Since we will repeatedly train and predict multiple times, we create train() and predict() functions

```
def train(df_train, y_train):
    dicts = df_train[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression()
    model.fit(X_train, y_train)

    return dv, model
```

```
dv, model = train(df_train, y_train)
```

train() takes the df_train, y_train. Convert df_train into dictionary. Create DictVectorizer dv to do One-hot encoding of dicts. Fit & Transform the dv on dicts to prepare the Feature Matrix X_train. Fit the model on X_train, y_train. Returns the dv & model

```
def predict(df, dv, model):
     dicts = df[categorical + numerical].to_dict(orient='records')

     X = dv.fit_transform(dicts)
     y_pred = model.predict_proba(X)[:,1]

     return y_pred
```

```
y_pred = predict(df_val, dv, model)
y_pred

# Output: array([0.00899722, 0.20451861, 0.2122173 , ..., 0.13639118,
```

predict() function takes in the df_val, dv, model and prepares the Validation Feature Matrix X. Use model.predict_proba(X) to get the Predicted Probability values in y_pred

Now, we implement K-fold CV using sklearn.model_selection import KFold. We initialize the KFold object with splits= 10. i.e. there will be 10 splits of df_full_train - 90% data will be training and 10% data will be Validation set.
kfold.split returns a generator object (like an iterator). So, if we use next(kfold.split()), it will return list of numbers that will be used as index for the Training and Validation set.
len(train_idx), len(val_idx) are the number of elements Out of 5634, 5070 are in Training set and 564 are in validation set (90:10). Using train_idx, val_idx, you can get subset of df_full_train as df_train and df_val

```
from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=1)

kfold.split(df_full_train)
# Output: <generator object _BaseKFold.split at 0x2838baf20>

train_idx, val_idx = next(kfold.split(df_full_train))
len(train_idx), len(val_idx)
# Output: (5070, 564)

len(df_full_train)
# Output: 5634

# We can use iloc to select a part of this dataframe
df_train = df_full_train.iloc[train_idx]
df_val = df_full_train.iloc[val_idx]
```

Now, replace the next() call with a for..in loop. And for each iteration, we compute auc_score and capture it in scores[]

```
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score

kfold = KFold(n_splits=10, shuffle=True, random_state=1)
scores = []

for train_idx, val_idx in kfold.split(df_full_train):
    df_train = df_full_train.iloc[train_idx]
    df_val = df_full_train.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)
```

→

```
scores
# Output:
# [0.8479398247539081,
# 0.8410581683168317,
# 0.8557214756739697,
# 0.8333552794008724,
# 0.8262717121588089,
# 0.8342657342657342,
# 0.8412569195701727,
# 0.8186669829222013,
# 0.8452349192233585,
# 0.8621054754462034]
```

The above is a long-running logic and takes time. So, to show the progress in the Jupyter notebook or Python script, we can use Python package called tqdm. Install tqdm using “pip install tqdm”.

```
from sklearn.model_selection import KFold
!pip3 install tqdm
from tqdm.auto import tqdm

kfold = KFold(n_splits=10, shuffle=True, random_state=1)
scores = []

for train_idx, val_idx in tqdm(kfold.split(df_full_train)):
    df_train = df_full_train.iloc[train_idx]
    df_val = df_full_train.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)
```

→

```
    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)

3/? [00:12<00:00, 4.33s/it]
```

```
scores
# Output:
# [0.8479398247539081,
# 0.8410581683168317,
# 0.8557214756739697,
# 0.8333552794008724,
# 0.8262717121588089,
# 0.8342657342657342,
# 0.8412569195701727,
# 0.8186669829222013,
# 0.8452349192233585,
# 0.8621054754462034]
```

Use the scores generated to compute the average score across the 10 folds, which is 84.1%, with a standard deviation of 0.012.

```
print('%.3f +- %.3f' % (np.mean(scores), np.std(scores)))
# Output: 0.841 +- 0.012
```

Logistic Regression Parameter Tuning (C)

Just like we have Regularization parameter , r for Linear Regression, to tune the model, similarly, we have Parameter C for Logistic Regression. In Linear Regression, Larger values of r will lead to Stronger Regularization. But, in Logistic Regression, Smaller values of C will lead to Stronger Regularization

model = LogisticRegression(…, C,…)
Default value of C = 1. C cannot be 0.0

To control it, we pass it to LogisticRegression Constructor in the train() function

```
def train(df_train, y_train, C=1.0):
    dicts = df_train[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression(C=C, max_iter=1000)
    model.fit(X_train, y_train)

    return dv, model
```

```
dv, model = train(df_train, y_train, C=0.001)
```

→ We are setting C = 0.001

So, we pass in different values of C using an outer for loop and see that the Mean AUC & Std Deviation is nearly the same for all values of C. So, lets keep the default value of C=1.0

```
from sklearn.model_selection import KFold

n_splits = 5

for C in tqdm([0.001, 0.01, 0.1, 0.5, 1, 5, 10]):
    scores = []

    kfold = KFold(n_splits=n_splits, shuffle=True, random_state=1)

    for train_idx, val_idx in kfold.split(df_full_train):
        df_train = df_full_train.iloc[train_idx]
        df_val = df_full_train.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        scores.append(auc)

    print('C=%s %.3f +- %.3f' % (C, np.mean(scores), np.std(scores)))
```

```
# Output:
#  14%|██          | 1/7 [00:01<00:06,  1.03s/it]
# C=0.001 0.825 +- 0.009
# 29%|████        | 2/7 [00:02<00:05,  1.07s/it]
# C=0.01 0.840 +- 0.009
# 43%|██████      | 3/7 [00:03<00:04,  1.06s/it]
# C=0.1 0.840 +- 0.008
# 57%|████████    | 4/7 [00:04<00:03,  1.08s/it]
# C=0.5 0.841 +- 0.006
# 71%|██████████  | 5/7 [00:05<00:02,  1.13s/it]
# C=1 0.841 +- 0.008
# 86%|████████████| 6/7 [00:06<00:01,  1.15s/it]
# C=5 0.841 +- 0.007
# 100%|██████████| 7/7 [00:07<00:00,  1.10s/it]
# C=10 0.841 +- 0.008
```

And, now that we have determined **to keep C=1.0 and the mean AUC is 84% which is GOOD**, we train using the FULL Train set and verify with the TEST Set.

```
dv, model = train(df_full_train, df_full_train.churn.values, C=1.0)
y_pred = predict(df_test, dv, model)

auc = roc_auc_score(y_test, y_pred)
auc
# Output: 0.8572386167896259
```

We see that the AUC is slightly better than what we observed during k-fold cross-validation but that is not significant

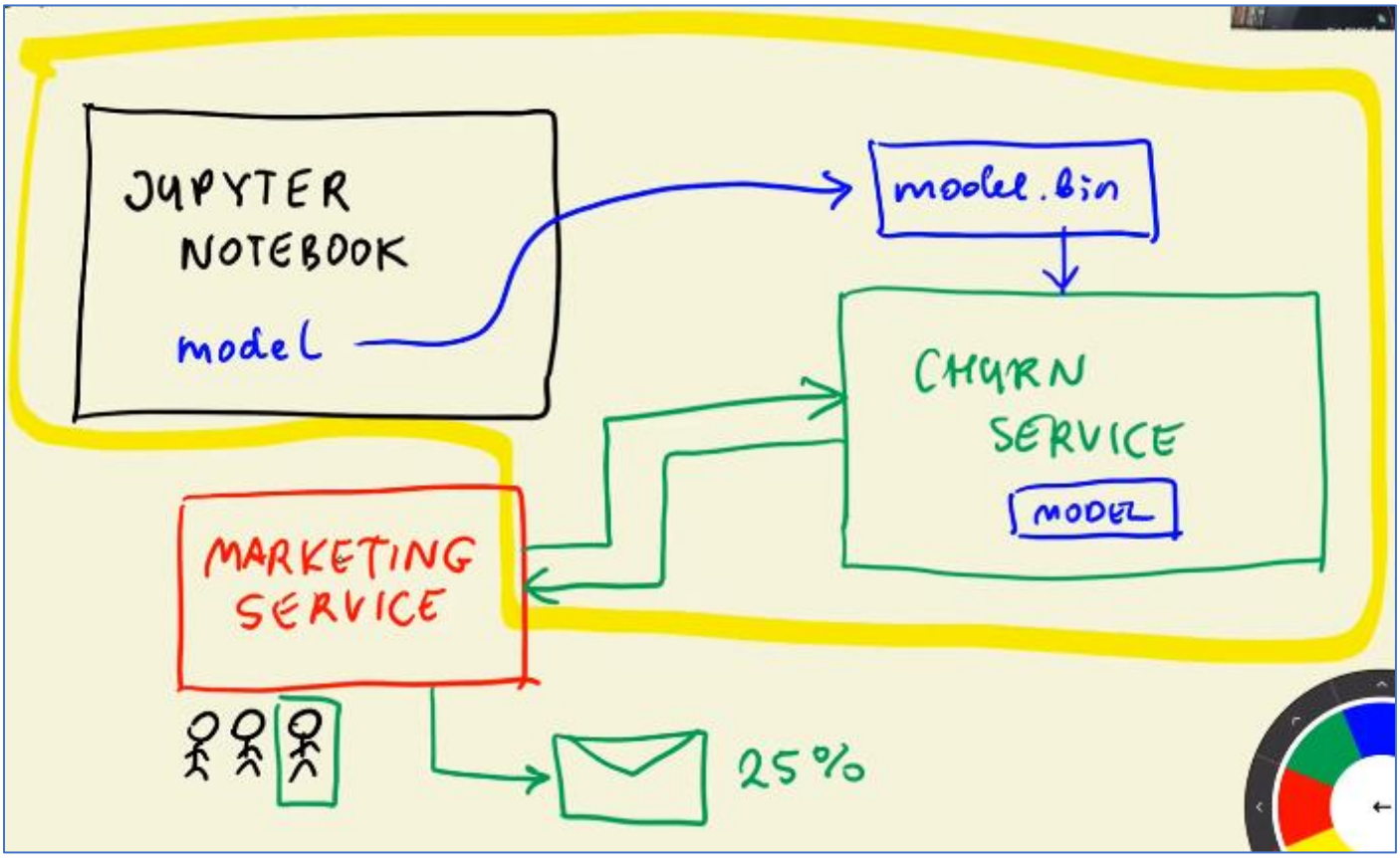
When to use HOLD OUT Strategy and when to use CROSS VALIDATION Strategy?

For larger datasets, standard hold-out validation is often sufficient. However, if your dataset is smaller or you require insight into the model's stability and variation across folds, then cross-validation is more appropriate. For larger datasets, consider using fewer splits (e.g., 2 or 3), while for smaller datasets, a higher number of splits (e.g., 10) may be beneficial.

5. DEPLOYING ML MODELS

INTRODUCTION

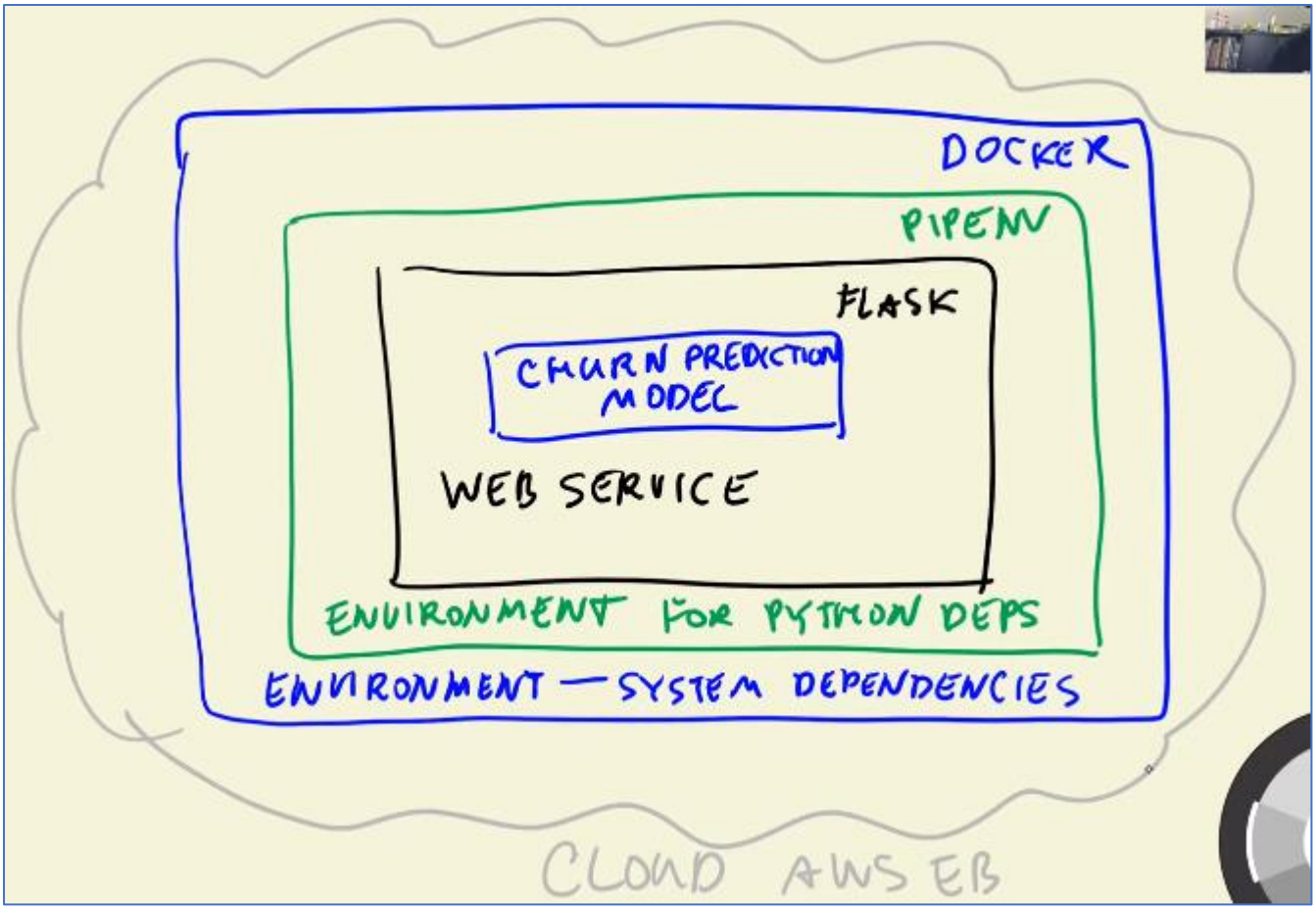
Once we have tested out and found the best model for the use case using Jupyter notebook, we want to deploy the model to use it.



So, for our Churn Prediction model, we have the model in our Jupyter notebook which we'll save as a model.bin file. This bin file will be hosted within a Python Web Service called CHURN Service. Assume there is a Marketing service which will send a user's profile to the Churn Service and our model will return back the Prediction of Churn/No Churn based on which the Marketing Service will either send the 25% promo email or not

ARCHITECTURE OF THE DEPLOYMENT

1. Model hosted within Webservice using Python FLASK framework
2. Webservice hosted alongwith Python Dependencies using PIPENV to isolate different services and their Python dependencies
3. Package the Webservice PIPENV alongwith system dependencies using DOCKER to run it anywhere
4. Host the Docker Container on AWS Elastic Beanstalk (or any other cloud or local environment) - optional



SAVING & LOADING MODEL

Training the model

Before we save the model, we need to train the model. Key code in Jupyter notebook is as follows (keeping only necessary statements that will go into the script) -

```
# Import statements

import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
```

```
# Parameters
C = 1.0         # Regularization
n_splits = 5    # No of folds for K-Fold
```

```
## Read the full dataset
df = pd.read_csv('3-BinaryClassification-ChurnPredictionProject-data.csv')

## Normalize the column names, types,
## replace missing values
df.columns = df.columns.str.lower().str.replace(' ', '_')

categorical_columns = list(df.dtypes[df.dtypes == 'object'].index)
for c in categorical_columns:
    df[c] = df[c].str.lower().str.replace(' ', '_')

df.totalcharges = pd.to_numeric(df.totalcharges, errors='coerce')
df.totalcharges = df.totalcharges.fillna(0)

df.churn = (df.churn == 'yes').astype(int)
```

```
### Full train, Test split
df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=1)
```

```
numerical = ['tenure', 'monthlycharges', 'totalcharges']

categorical = [
    'gender',
    'seniorcitizen',
    'partner',
    'dependents',
    'phoneservice',
    'multiplelines',
    'internetservice',
    'onlinesecurity',
    'onlinebackup',
    'deviceprotection',
    'techsupport',
    'streamingtv',
    'streamingmovies',
    'contract',
    'paperlessbilling',
    'paymentmethod',
]
```

```
### Train() function
def train(df_train, y_train, C=1.0):
    dicts = df_train[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    X_train = dv.fit_transform(dicts)

    model = LogisticRegression(C=C, max_iter=1000)
    model.fit(X_train, y_train)

    return dv, model
```

```
### Predict function
def predict(df, dv, model):
    dicts = df[categorical + numerical].to_dict(orient='records')

    X = dv.transform(dicts)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred
```

```
### KFold CV
kfold = KFold(n_splits=n_splits, shuffle=True, random_state=1)

scores = []

for train_idx, val_idx in kfold.split(df_full_train):
    df_train = df_full_train.iloc[train_idx]
    df_val = df_full_train.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train, C=C)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    scores.append(auc)

print('C=%s %.3f +- %.3f' % (C, np.mean(scores), np.std(scores)))
```

```
scores
```
✓ 0.0s

```
[0.842398599587098,
 0.8455854357038802,
 0.8311820188641381,
 0.8301724275756219,
 0.8522245928900882]
```

```
## Train the model
dv, model = train(df_full_train, df_full_train.churn.values, C=1.0)

## Validate using Test Dataset
y_pred = predict(df_test, dv, model)
y_test = df_test.churn.values

## Check the AUC
auc = roc_auc_score(y_test, y_pred)
auc
```

Save the model

We'll use Pickle which is built-in Python library to save Python objects

```
import pickle
```

Name our model file before we can write it to a file

```
output_file = f'model_C={C}.bin'
output_file
# Output: 'model_C=1.0.bin'
```

Open the file in ‘wb’ mode means Write Binary. We need to save DictVectorizer and the model as well, because with just the model we’ll not be able to translate a customer into a feature matrix. Closing the file is crucial. Otherwise, we cannot be certain whether this file truly contains the content. To avoid accidentally forgetting to close the file, we can use the ‘with’ statement, which ensures that the file is closed automatically. Everything we do inside the ‘with’ statement keeps the file open. However, once we exit this statement, the file is automatically closed.

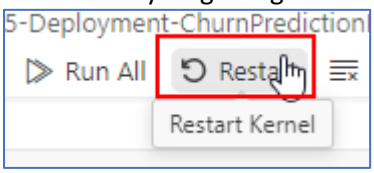
```
f_out = open(output_file, 'wb')

pickle.dump((dv, model), f_out)

f_out.close()
```

→ Instead, use this

```
with open(output_file, 'wb') as f_out:
    pickle.dump((dv, model), f_out)
```

Load the model

Since we have the Save model and Load model code in the same Jupyter notebook, if we want to truly simulate Load the model by forgetting the execution done till now, you need to “Restart Kernel”



Import pickle and specify filename

```
import pickle


model_file = 'model_C=1.0.bin'
```

Open the file using ‘with’ statement in Read Binary mode. Here, ‘rb’ denotes Read Binary. We employ the ‘load’ function from pickle, which returns both the DictVectorizer and the model.

```
with open(model_file, 'rb') as f_in:
    dv, model = pickle.load(f_in)

dv, model
# Output: (DictVectorizer(sparse=False), LogisticRegression(max_iter=1000))
```

Model is now loaded. Now, simulate a customer record whose probability score we will predict using the model

```
customer = {
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'yes',
    'dependents': 'no',
    'phoneservice': 'no',
    'multiplelines': 'no_phone_service',
    'internetservice': 'dsl',
    'onlinesecurity': 'no',
    'onlinebackup': 'yes',
    'deviceprotection': 'no',
    'techsupport': 'no',
    'streamingtv': 'no',
    'streamingmovies': 'no',
    'contract': 'month-to-month',
    'paperlessbilling': 'yes',
    'paymentmethod': 'electronic_check',
    'tenure': 1,
    'monthlycharges': 29.85,
    'totalcharges': 29.85
}
```

Before we can apply the predict function to this customer we need to turn it into a feature matrix. The DictVectorizer expects a list of dictionaries, that's why we create a list with one customer.

```
X = dv.transform([customer])
X

# Output:
# array([[ 1.  ,  0.  ,  0.  ,  1.  ,  0.  ,  1.  ,  0.  ,  0.  ,  1.  ,
#         0.  ,  1.  ,  0.  ,  0.  , 29.85,  0.  ,  1.  ,  0.  ,  0.  ,
#         0.  ,  1.  ,  1.  ,  0.  ,  0.  ,  0.  ,  1.  ,  0.  ,  1.  ,
#         0.  ,  0.  ,  1.  ,  0.  ,  1.  ,  0.  ,  0.  ,  1.  ,  0.  ,
#         0.  ,  1.  ,  0.  ,  0.  ,  1.  ,  0.  ,  0.  ,  1.  , 29.85]])
```

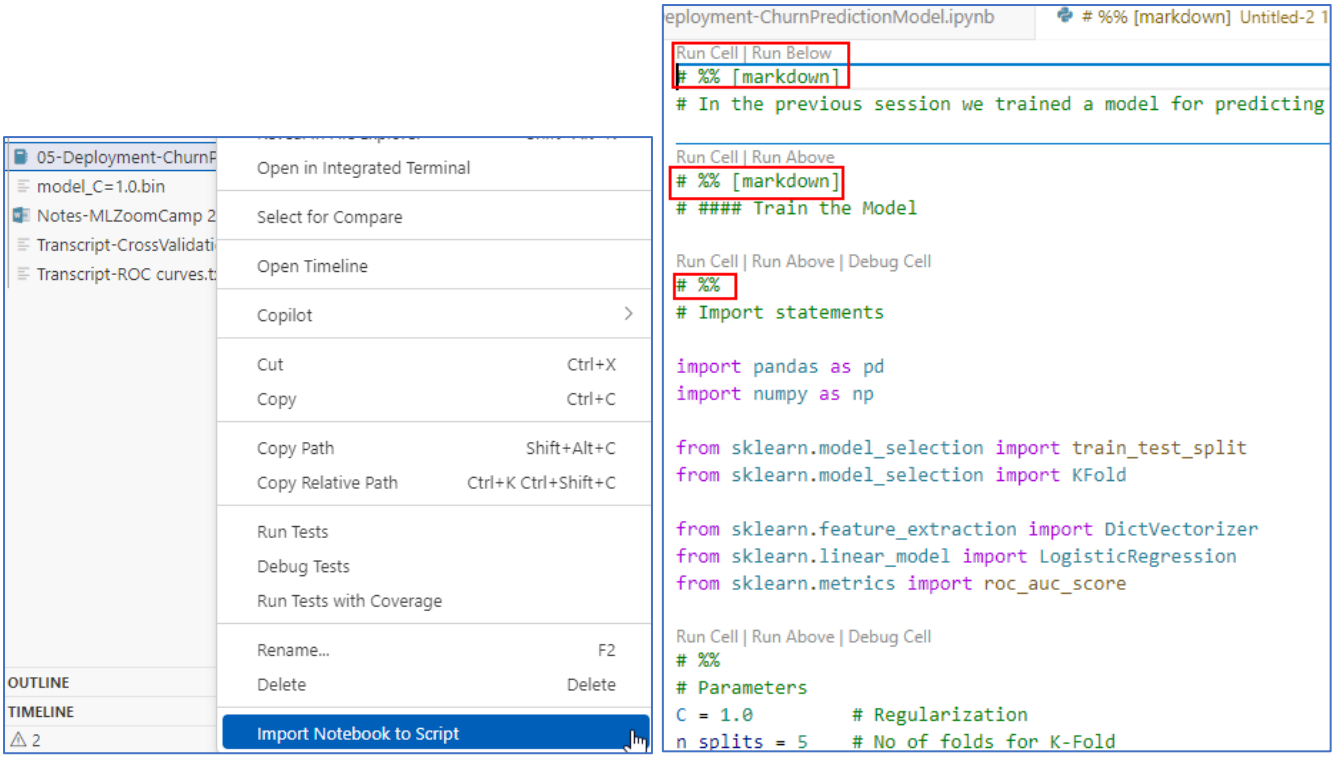
We use predict function to get the probability that this particular customer is going to churn. We're interested in the second element, so we need to set the row=0 and column=1.

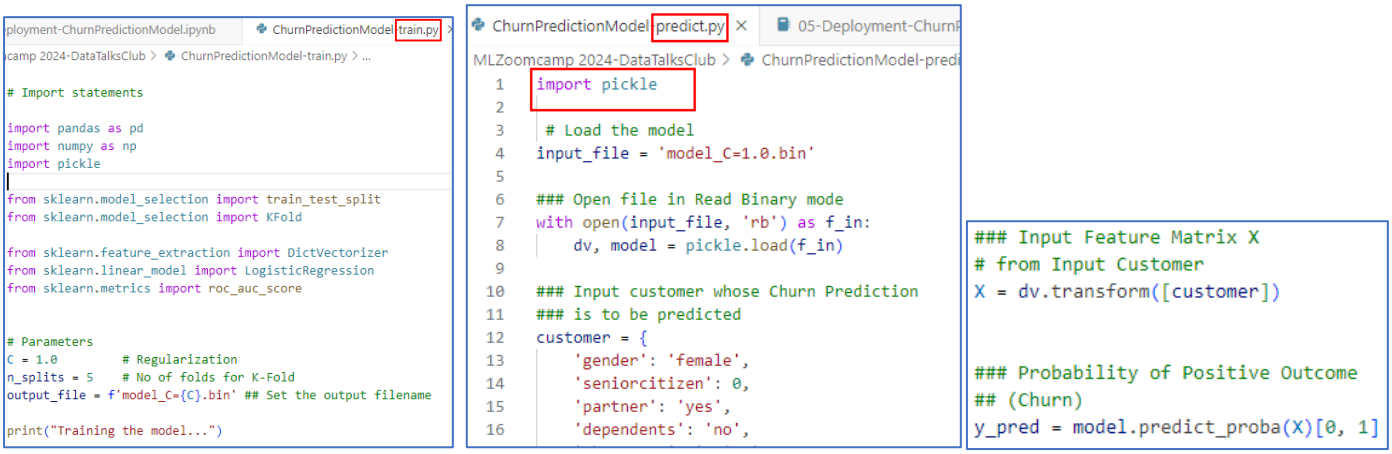
```
model.predict_proba(X)
# Output: array([[0.36364158, 0.63635842]])

model.predict_proba(X)[0,1]
# Output: 0.6363584152758612
```

Create Python script from Jupyter Notebook

Depending on the IDE being used to access the Jupyter notebook, the steps varies but all are very simple to use

1. In VSCode Desktop, right-click on the .ipynb file -> Import Notebook to Script. This will open a new .py file with the contents of the Notebook. But, it will have all these "# %%" statements
2. Highlight any one of the "# %%" word → Hit Ctrl+F2 (Change all occurences) → Edit any of those words to blank "". Repeat for any other unwanted text
3. Save the file as ChurnPredictionModel-train.py script
4. Edit the ChurnPredictionModel-train.py file to do simple re-org/cleanup of code statements and adding some print statements to log the progress. Keep the code of Training+Saving Model in this file
5. Move the code for Loading the model & Predicting to separate file ChurnPredictionModel-predict.py file

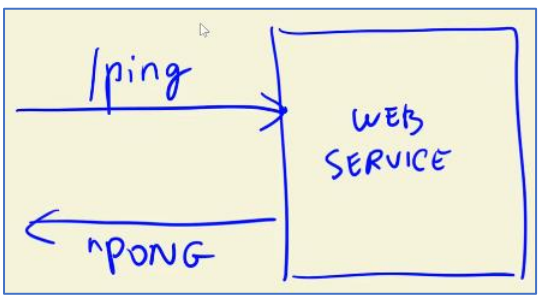




Note: In Predict.py, you would expect that to call dv.transform() & model.predict_proba(), you would need to reference all the imports or the train.py module. But, you don't need to do that because Pickle serializes and saves the (dv, model) objects to the .bin file which contains all the references. (TODO-Check how does this actually work??)

Web Services: Intro to Flask

A **web service** is a method for communicating between two devices over a network. So, let's say we have our web service and a user who wants to make a request. So, the user sends the request with some information. The request has some parameters, then the user gets back result with the answer of this request.
We use **Flask** for implementing the web service and it takes care of all the internals.



To understand how to use Flask, we build a very simple Webservice. User will send request on http://localhost:port/ping URL and Webservice will send response as "PONG" string

We install Flask using >>> pip install flask

ping.py-

```python
from flask import Flask

app = Flask('ping')

@app.route('/ping', methods=['GET'])
def ping():
    return "PONG"

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

We create a Python script ping.py

- Import Flask
- Create app object of type Flask and name “ping”
- Define the ping() function which simply returns “PONG” string
- Use @app.route Decorator to specify the URL route /ping and the GET method. So, when you send GET request to http://localhost:9696/ping route, this ping() function will be executed
- At the end, add the main code to run this app when this .py file is directly executed in terminal (If this .py file is used a library, then the code in the if loop is not run)

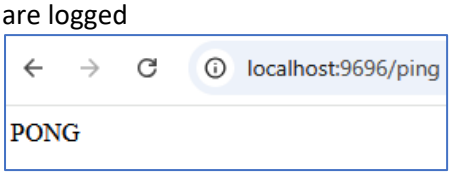
Run the ping.py (>>> python ping.py) in one terminal to start the webservice.

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> .\5-ping.py
 * Serving Flask app 'ping'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:9696
 * Running on http://192.168.29.174:9696
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 300-566-151
```

In another terminal, send request through Curl command (works in Windows too). You get the HTTP Response as below-

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> curl http://localhost:9696/ping


StatusCode        : 200
StatusDescription : OK
Content           : PONG
RawContent        : HTTP/1.1 200 OK
                    Connection: close
                    Content-Length: 4
                    Content-Type: text/html; charset=utf-8
                    Date: Thu, 10 Apr 2025 11:11:59 GMT
                    Server: Werkzeug/3.0.6 Python/3.8.10

                    PONG
Forms             : {}
Headers           : {[Connection, close], [Content-Length, 4], [Content-Type, text/html; charset=utf-8], [Date, Thu, 10 Apr 2025 11:11:59 GMT]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 4
```

In browser, send request and you get the string “PONG”. Also, in the Terminal running the Webservice, the requests are logged




```
 * Debugger PIN: 300-566-151
127.0.0.1 - - [10/Apr/2025 16:41:59] "GET /ping HTTP/1.1" 200 -
127.0.0.1 - - [10/Apr/2025 16:42:59] "GET /ping HTTP/1.1" 200 -
127.0.0.1 - - [10/Apr/2025 16:42:59] "GET /favicon.ico HTTP/1.1" 404 -
```

Serving the model through web service

We modify the predict.py as predict-service.py to wrap up the predict() function inside Flask Webservice at http://localhost:9696/predict . You need additional import statements for supporting classes/functions - Flask.request, Flask.jsonify.

predict-service.py

ncamp 2024-DataTalksClub > 5-ChurnPredictionModel-predict-service.py

```
import pickle
from flask import Flask
from flask import request
from flask import jsonify

# Load the model
input_file = 'model_C=1.0.bin'

### Open file in Read Binary mode
with open(input_file, 'rb') as f_in:
    dv, model = pickle.load(f_in)
```

```
app = Flask('Churn')

@app.route('/predict', methods=['POST'])
def predict():
    customer = request.get_json()

    ### CORE LOGIC - BEGIN
    X = dv.transform([customer]) ### Input Feature Matrix X
    y_pred = model.predict_proba(X)[0, 1] ### Probability of Positive Outcome
    churn = y_pred >= 0.5
    ### CORE LOGIC - END

    result = {
        #'Churn_probability' : y_pred,
        ##above line generates error in jsonify
        # as it is a Numpy float64. Needs to be
        # converted to Python float
        'Churn_probability' : float(y_pred),
        #'Churn' : churn
        ##above line generates error in jsonify
        # as it is a Numpy Boolean. Needs to be
        # converted to Python boolean
        'Churn' : bool(churn)
    }

    return jsonify(result)

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Customer is read from request body as JSON using request.get_json(). Then, the core logic is to convert the Customer to Input Feature Matrix X, predict probability and make decision with threshold = 0.5.
Result is JSON with Churn Probability & Churn Decision.

And then we create another script for testing (will send request to service & get response)

predict-service-tester.py

```
ncamp 2024-DataTalksClub > 5-ChurnPredictionModel-predict-service-tester.py > ...
import requests

url = 'http://localhost:9696/predict'

customer = {
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "yes",
    "dependents": "no",
    "phoneservice": "no",
    "multiplelines": "no_phone_service",
    "internetservice": "dsl",
    "onlinesecurity": "no",
    "onlinebackup": "yes",
    "deviceprotection": "no",
    "techsupport": "no",
    "streamingtv": "no",
    "streamingmovies": "no",
    "contract": "monthly",
    "paperlessbilling": "yes",
    "paymentmethod": "electronic_check",
    "tenure": 1,
    "monthlycharges": 29.85,
    "totalcharges": 29.85
}

response = requests.post(url, json=customer).json()

if response['Churn']:
    print(f"send promo email to customer with Probability {response['Churn_probability']}")
else:
    print(f"Dont send promo email to customer with Probability {response['Churn_probability']}")
```

We import “requests” class which is a Python class (different from Flask.request) to send Post request. We define Customer as JSON (which is same as Python dict). requests.post() will get response and we do .json() on it to convert the response to JSON. If the response[Churn] value is True, send promo email else don’t send

Running the webservice

Run the predict-service.py in one terminal window

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> .\5-ChurnPredictionModel-predict-service.py
 * Serving Flask app 'Churn'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:9696
 * Running on http://192.168.29.174:9696
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 300-566-151
```

And run the predict-service-tester.py in another window

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> .\5-ChurnPredictionModel-predict-service-tester.py
send promo email to customer with Probability 0.5724415801097334
```

You can change the values of Customer in the tester file to test for other values

Preparing for Production: Gunicorn on Linux, Unix, MacOS, Windows Subsystem for Linux (WSL)

When you start a Flask app, it shows warning that it is a Development Server and not to be used for Production. So, there are many Production WSGI (Web Server Gateway Interface) servers in Python. One of them is Gunicorn but it works only for Linux, Unix, MacOS, WSL since it uses some Unix-only components.

First, you install gunicorn using >>> pip install gunicorn

Then, >>> gunicorn --bind 0.0.0.0:9696 <.py filename>:app
This command binds the web service to port 9696 on localhost and uses the 'predict' function from our app which we've implemented in 'predict-service.py'

```
$ gunicorn --bind 0.0.0.0:9696 predict:app
[2021-10-01 06:54:19 +0200] [5112] [INFO] Starting gunicorn 20.1.0
[2021-10-01 06:54:19 +0200] [5112] [INFO] Listening at: http://0.0.0.0:969
6 (5112)
[2021-10-01 06:54:19 +0200] [5112] [INFO] Using worker: sync
[2021-10-01 06:54:19 +0200] [5146] [INFO] Booting worker with pid: 5146
```

gunicorn will not use the below if statement of the predict-service.py file. So, we bind and run it using the above command

```
if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Preparing for Production: Waitress for Windows

If you are using native Windows for development, you can use Waitress as alternative to Gunicorn

First, install waitress using >>> pip install waitress

Then, run command waitress-serve to listen on mentioned host:port
>>> waitress-serve --listen 0.0.0.0:9096 <.py filename>:app

NOTE- For waitress-serve command, the .py filename should not contain hyphen/underscores else it gives error

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> waitress-serve --listen 0.0.0.0:9696 5-ChurnPredictionMode
l-predict-service:app
Error: Malformed application '5-ChurnPredictionModel-predict-service:app'
```

So, for this demo, temporarily rename .py file to predict.py and serve using waitress

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> waitress-serve --listen 0.0.0.0:9696 predict:app
INFO:waitress:Serving on http://0.0.0.0:9696
```

And then execute the tester. It works

```
PS D:\Data\Celsius Laptop_20211111\Study\MachineLearning\MLZoomcamp 2024-DataTalksClub> .\5-ChurnPredictionModel-predict-service-tester.py
send promo email to customer with Probability 0.5724415801097334
```

Configure Windows Subsystem for Linux

Installed WSL With Ubuntu distro using powershell >> wsl --install

In Start Menu, type "wsl" and click the "WSL" app to start Linux

Username: jg_linux, Pwd: Newuser123

~ : Home directory /home/jg_linux
/mnt : Windows Filesystem /mnt/c- C:\ drive; /mnt/d - D:\ drive (external HDD)

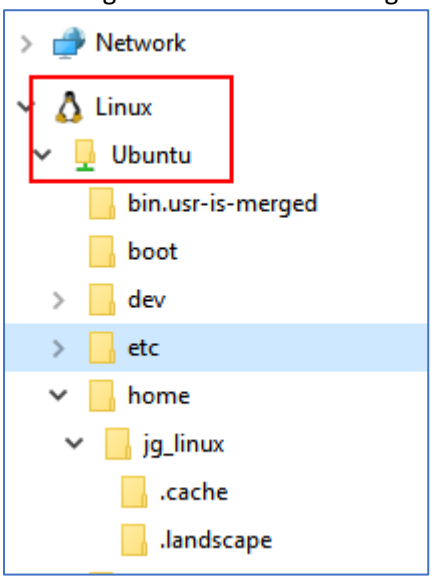
For ML Zoomcamp files in external HDD, the path is very long and occupies lot of space of Command prompt

```
jg_linux@JG-LifebookE752:/mnt/d/data/Celsius Laptop_20211111/study/MachineLearning$ cd 'MLZoomcamp 2024-DataTalksClub'
jg_linux@JG-LifebookE752:/mnt/d/data/Celsius Laptop_20211111/study/MachineLearning/MLZoomcamp 2024-DataTalksClub$ ls -altotal 33220
drwxrwxrwx 1 root root     4096 Apr 10 15:05  .
drwxrwxrwx 1 root root     4096 Mar 26 10:36  ..
-rwxrwxrwx 1 root root    15992 Mar 28 11:11  1.7-numpy.ipynb
-rwxrwxrwx 1 root root    10933 Mar 30 06:41  1.8-linear-algebra.ipynb
-rwxrwxrwx 1 root root    57268 Mar 30 07:14  1.9-pandas.ipynb
-rwxrwxrwx 1 root root  1475504 Mar 31 05:54  2-Regression-CarPricePrediction-data.csv
-rwxrwxrwx 1 root root   249045 Apr  2 09:49  2-Regression-CarPricePrediction.ipynb
-rwxrwxrwx 1 root root   977501 Apr  2 09:57  3-BinaryClassification-ChurnPredictionProject-data.csv
-rwxrwxrwx 1 root root   215429 Apr  4 12:08  3-BinaryClassification-ChurnPredictionProject.ipynb
-rwxrwxrwx 1 root root   168687 Apr  7 09:14  4-EvaluationMetrics.ipynb
-rwxrwxrwx 1 root root      908 Apr 10 12:19  5-ChurnPredictionModel-predict-service-tester.py
```

So, created symbolic link (shortcut) in home directory (~ or /home/jg_linux) to point to the Zoomcamp folder
ln -s '/mnt/d/data/Celsius Laptop_20211111/study/MachineLearning/MLZoomcamp 2024-DataTalksClub' ~/MLZC

```
jg_linux@JG-LifebookE752:~$ ln -s '/mnt/d/data/Celsius Laptop_20211111/study/MachineLearning/MLZoomcamp 2024-DataTalksClub' ~/MLZC
jg_linux@JG-LifebookE752:~$ cd MLZC
jg_linux@JG-LifebookE752:~/MLZC$ ls -al
total 33228
drwxrwxrwx 1 root root     4096 Apr 11 10:34  .
drwxrwxrwx 1 root root     4096 Mar 26 10:36  ..
-rwxrwxrwx 1 root root    15992 Mar 28 11:11  1.7-numpy.ipynb
-rwxrwxrwx 1 root root    10933 Mar 30 06:41  1.8-linear-algebra.ipynb
-rwxrwxrwx 1 root root    57268 Mar 30 07:14  1.9-pandas.ipynb
-rwxrwxrwx 1 root root  1475504 Mar 31 05:54  2-Regression-CarPricePrediction-data.csv
-rwxrwxrwx 1 root root   249045 Apr  2 09:49  2-Regression-CarPricePrediction.ipynb
-rwxrwxrwx 1 root root   977501 Apr  2 09:57  3-BinaryClassification-ChurnPredictionProject-data.csv
-rwxrwxrwx 1 root root   215429 Apr  4 12:08  3-BinaryClassification-ChurnPredictionProject.ipynb
```

Accessing files within WSL through Explorer. It will be shown as Network drive “Linux”



<u>To shutdown/terminate the Linux VM-</u>

If you simply close the WSL window, the VM still runs in background. To completely stop the VM, do the following

PS>> wsl --list --verbose

```
PS C:\Windows\system32> wsl --list --verbose
  NAME      STATE           VERSION
* Ubuntu    Running         2
```

PS>> wsl -t <DISTRO_NAME> → wsl -t Ubuntu

```
PS C:\Windows\system32> wsl -t Ubuntu
The operation completed successfully.
```

Install Windows Terminal app from Microsoft Store to have all Powershell, Cmd, WSL shells as tabs in one window -> Good for switching between shells

Software programs in Linux are called PACKAGES. They are installed using APT (Advanced Package Tool) apt-get command.

Python Virtual Environment

Dependency and environment management

When you run 'pip install scikit-learn,' it searches in the directories listed in the $PATH variable, such as ~/usr/bin/ in this case. Inside this folder, you have 'pip,' 'python,' and other packages. The 'pip' from this folder is used, and it connects to the Python Package Index (PyPI) at pypi.org. It then installs the latest version of the scikit-learn package into the directory specified in the $PATH variable.

Now, assume you have two applications

1. Churn service, which relies on scikit-learn==0.24.2.
2. Lead scoring service, which depends on scikit-learn==1.0.

Here, we need two different versions of scikit-learn to be installed side-by-side and they need to be isolated from each other

Virtual environments

On the same physical machine, we create two separate virtual environments ~/venv/churn & ~/venv/lead, each with its own Python installation for the services:

1. Churn service
   Here, the Python resides in ~/venv/churn/bin/python while its PIP resides in ~/venv/churn/bin/pip
   Here, we run >> pip install scikit-learn==0.24.2

2. Lead Scoring Service
   Here, the Python resides in ~/venv/lead/bin/python while its PIP resides in ~/venv/lead/bin/pip
   Here, we run >> pip install scikit-learn==1.0

This approach ensures that you won't encounter conflicts when using different versions of scikit-learn for the two services.

There are multiple options for managing Virtual Environments in Python-

1. **Venv:** Virtual environments, often referred to as "venv" or "virtualenv," allow you to create isolated Python environments for your projects. This separation helps manage dependencies and avoids conflicts between packages used in different projects.
2. **Conda:** Conda is an open-source package management and environment management system that can handle both Python and non-Python packages. It's commonly used in data science and scientific computing for managing complex environments.
3. **Poetry:** Poetry is a dependency management and packaging tool for Python. It simplifies the process of managing project dependencies, packaging, and publishing Python packages. Poetry aims to provide a consistent and user-friendly approach to Python development.
4. **Pipenv:** Pipenv is another tool for managing dependencies and virtual environments. It combines pip (Python's package installer) and virtualenv into one tool, making it easier to create and manage virtual environments and package dependencies for your projects.

Each of these tools has its strengths and may be more suitable for specific use cases. The choice of which one to use depends on your project's requirements and your personal preferences.

Starting with Python 3.12, installing packages via pip without virtual environments is discouraged. If you do pip install <>, you would get error

```
jg_linux@JG-LifebookE752:/usr/bin$ pip install pandas
error: externally-managed-environment

× This environment is externally managed
╰─> To install Python packages system-wide, try apt install
    python3-xyz, where xyz is the package you are trying to
    install.

    If you wish to install a non-Debian-packaged Python package,
    create a virtual environment using python3 -m venv path/to/venv.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip. Make
    sure you have python3-full installed.

    If you wish to install a non-Debian packaged Python application,
    it may be easiest to use pipx install xyz, which will manage a
    virtual environment for you. Make sure you have pipx installed.

    See /usr/share/doc/python3.12/README.venv for more information.

note: If you believe this is a mistake, please contact your Python installation or OS distribution provider. You can override this, at
the risk of breaking your Python installation or OS, by passing --break-system-packages.
hint: See PEP 668 for the detailed specification.
```

This is because Linux uses many Python packages of specific versions for its purposes. If you update the system-wide packages, there is a chance of breaking the OS. The error above also gives the option to circumvent this message only if you are willing to take the risk of breaking the OS.
So, developing Python programs with Virtual Environments is the BEST PRACTICE

Alexey was using Python 3.8 in Oct 2021

Alexey: Pipenv

First, in the project working directory, install pipenv using pip
>>> pip install pipenv

```
$ pip install pipenv
```

Then, we use pipenv to install libraries (replace pip with pipenv in the commands)
>>> pipenv install numpy scikit-learn==0.24.2 flask gunicorn

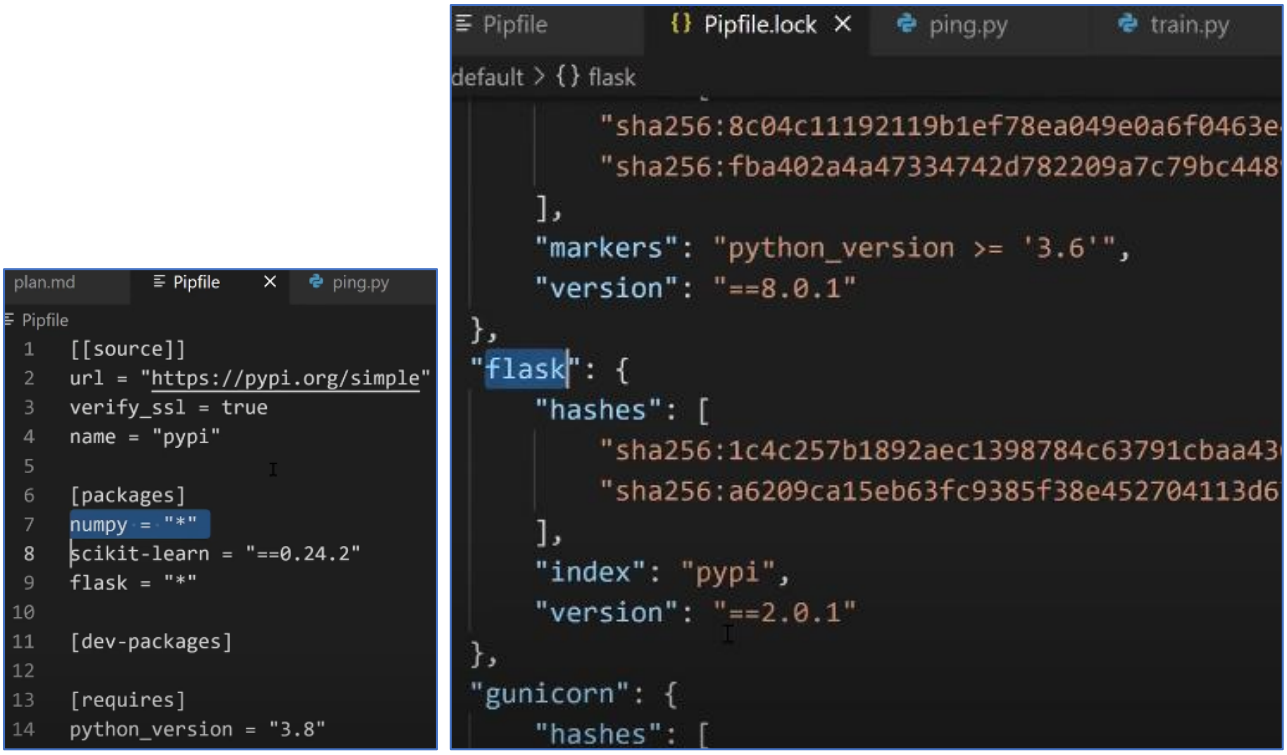
```
$ pipenv install numpy scikit-learn==0.24.2 flask
Creating a Pipfile for this project...
Installing numpy...
Adding numpy to Pipfile's [packages]...
✓ Installation Succeeded
Installing scikit-learn==0.24.2...
Adding scikit-learn to Pipfile's [packages]...
✓ Installation Succeeded
Installing flask...
Adding flask to Pipfile's [packages]...
✓ Installation Succeeded
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Building requirements...
Resolving dependencies...
⠧ Locking...
```

```
✓ Success!
Updated Pipfile.lock (cf3fe4)!
Installing dependencies from Pipfile.lock (cf3fe4)...
  🐍                                0/0 —
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
$ ls
 05-train-churn-model.py   data-week-3.csv    predict-test.py
 Pipfile                  'model_C=1.0.bin'   predict.py
 Pipfile.lock              ping.py            prep
 __pycache__               plan.md            train.py
```

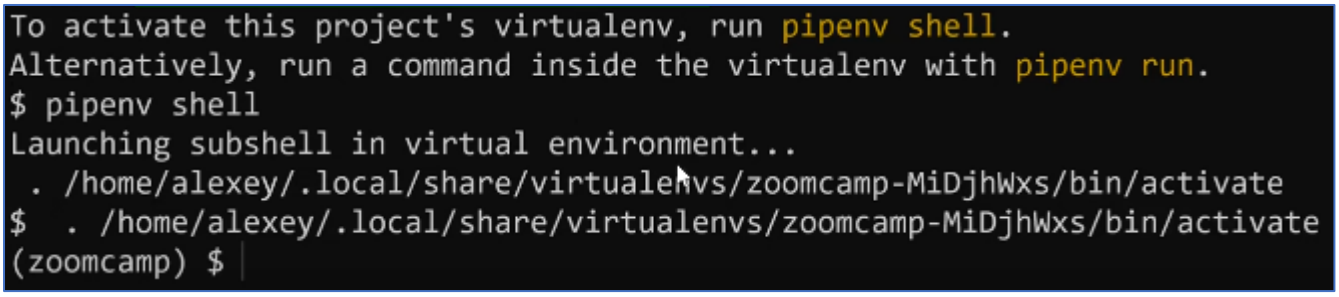
This will create a Virtual environment in the current project directory where our files are located. It will also install the libraries that will be used only by this VEnv. For this, it creates two files - Pipfile and Pipfile.lock

Pipfile and Pipfile.lock contents are as follows. Pipfile has the list of packages and from where they were downloaded. [dev-packages] are packages that you want only on your development environment but not when you deploy to production. Pipfile.lock contains the exact version/hash of the packages for security reasons

So, if you want to replicate your Pipenv environment on another system, you just need to copy the Pipfile, Pipfile.lock to the directory where you want to create your Pipenv environment and
run >>> pipenv install.
This will refer to the Pipfile/Pipfile.lock and download all the packages/dependencies as required by the Venv

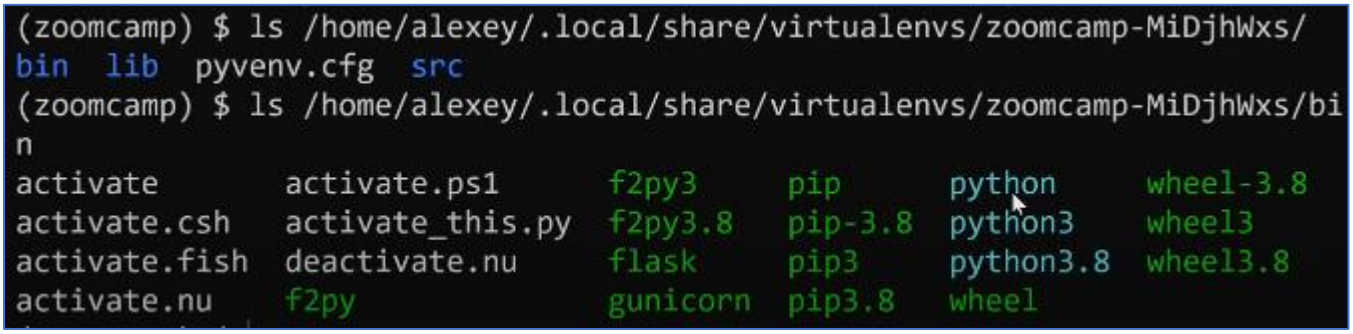


Activating the Pipenv



Ensure you are in the project directory where you have the Virtual Environment created. In that, run the cmd
>>> pipenv shell
Inside the pipenv shell, it tells you the location of where the actual VEnv files are stored on disk.
You can exit the environment with Ctrl+C

venv >>> ls ….path to virtual environment directory



This shows the python 3.8, pip, and other libraries that we installed

So, from within the VEnv, we run the Predict Web service
venv>>> gunicorn --bind 0.0.0.0:9696 predict:app

```
(zoomcamp) $ gunicorn --bind 0.0.0.0:9696 predict:app
[2021-10-01 10:18:24 +0200] [5732] [INFO] Starting gunicorn 20.1.0
[2021-10-01 10:18:24 +0200] [5732] [INFO] Listening at: http://0.0.0.0:969
6 (5732)
[2021-10-01 10:18:24 +0200] [5732] [INFO] Using worker: sync
[2021-10-01 10:18:24 +0200] [5734] [INFO] Booting worker with pid: 5734
```

And while the Service is running, we run the predict-tester from another terminal which connects to webservice

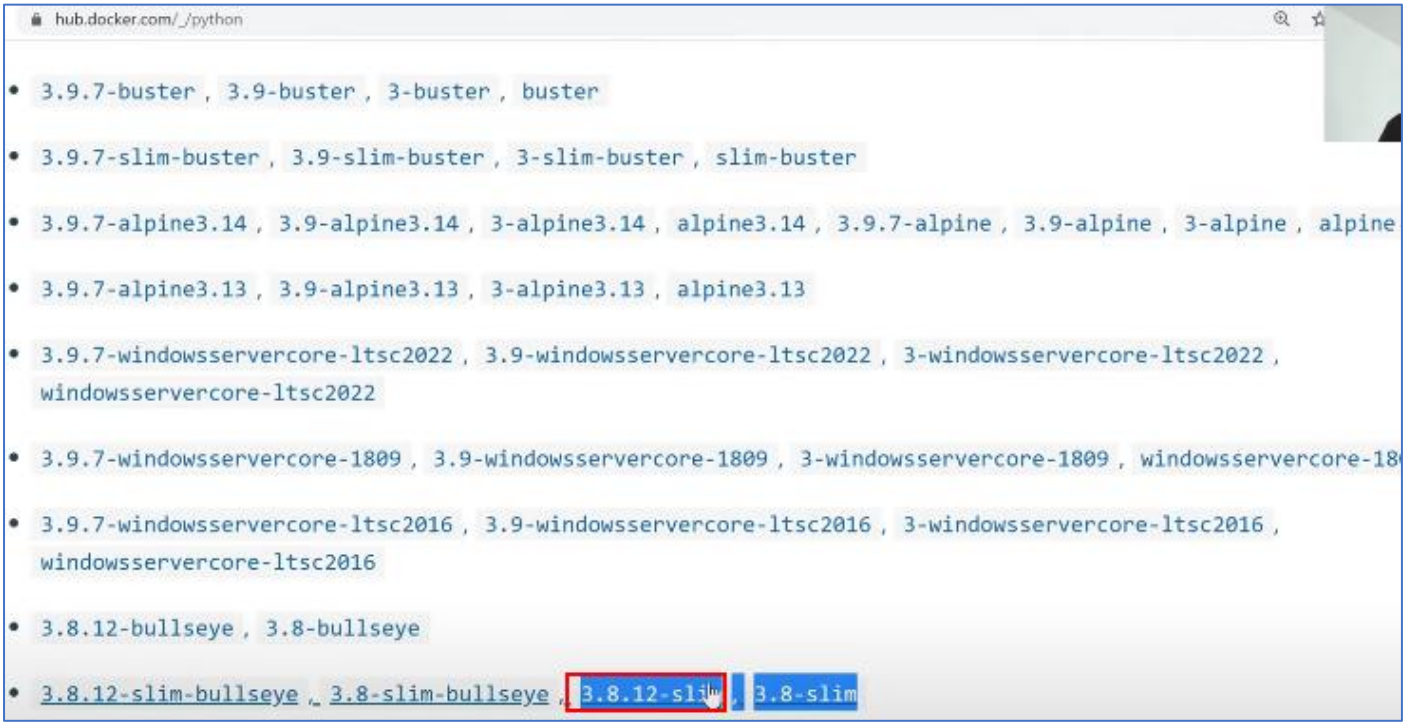
```
$ python predict-test.py
{'churn': False, 'churn_probability': 0.3257561103397851}
not sending promo email to xyz-123
```

ENVIRONMENT MANAGEMENT VIA DOCKER

Virtual environments are one step towards isolating services from each other within the Python environment. Using Docker is 2nd step towards isolating services from one other within the Host environment. So, for example, if our Churn Service requires Ubuntu 18.04 while our Lead Service requires Ubuntu 20.04, we can package each Service along with its OS dependencies as Docker images and run them on any machine.

Alexey: Docker

You search for Docker images of Python in Dockerhub



As a sample, we run the image “3.8.12-slim”. If the image is not available locally, Docker will download it and run the image

>>> docker run -it --rm python:3.8.12-slim

```
$ docker run -it --rm python:3.8.12-slim
Python 3.8.12 (default, Sep 28 2021, 19:14:48)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

-it = Interactive mode

--rm = Remove the image after we stop the container

This runs the Python interactive shell

Now, to get access to the terminal of the container of python:3.8.12-slim, we use additional switch entrypoint=bash
>>> docker run -it --rm --entrypoint=bash python:3.8.12-slim

```
$ docker run -it --rm --entrypoint=bash python:3.8.12-slim
root@6069600baca4:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

The ls command here shows the directory structure within the container

Then, inside the container we install the pipenv. Now, if we do this manually, we will need to manually do this everytime this container is started
container >>> pip install pipenv

```
root@6069600baca4:/test# pip install pipenv
Collecting pipenv
  Downloading pipenv-2021.5.29-py2.py3-none-any.whl (3.9 MB)
     |                                        | 3.9 MB 22.5 MB/s
Collecting virtualenv-clone>=0.2.5
  Downloading virtualenv_clone-0.5.7-py3-none-any.whl (6.6 kB)
Requirement already satisfied: pip>=18.0 in /usr/local/lib/python3
```

So, instead we programmatically configure the container using Dockerfile

Creating Dockerfile

A Dockerfile is a script-like text file used in Docker to build container images. It contains a series of instructions that Docker follows to create a reproducible and self-contained environment for running applications.
Here are some key aspects of a Dockerfile:

1. **Base Image:** You start by specifying a base image, which serves as the foundation for your container. Base images are typically lightweight Linux distributions or specific application images.
2. **Instructions:** Dockerfiles consist of various instructions, each responsible for a specific task. Common instructions include FROM (to set the base image), RUN (to execute commands inside the container during build), COPY (to copy files into the container), and CMD (to specify the default command to run when the container starts).
3. **Layering:** Docker images are built in layers. Each instruction in the Dockerfile creates a new layer. Layers are cached, and if nothing changes in a layer, Docker can reuse it from cache, making builds faster.
4. **Environment Configuration:** You can set environment variables, configure ports, define working directories, and perform other setup tasks to configure the container environment.
5. **File Copying:** You can copy files from the host system into the container image using the COPY instruction. This is often used to add application code, configuration files, or other assets.
6. **Container Execution Command:** The CMD instruction specifies the default command to run when the container is started. It's often used to launch the primary application within the container.
7. **Best Practices:** Following best practices in Dockerfile design can help create efficient and secure images. This includes minimizing the number of layers, avoiding installing unnecessary packages, and keeping images small.

Lets incrementally build the Dockerfile.

Version 1-

```
Dockerfile
1   FROM python:3.8.12-slim
2
3   RUN pip install pipenv
4
5   WORKDIR /app
6   COPY ["Pipfile", "Pipfile.lock", "./"]
```

1. FROM is the base image of this image

RUN pip install pipenv - Installs pipenv inside the container
WORKDIR /app - Creates '/app' directory in the container and switches to it
COPY ["Pipfile","Pipfile.lock", "./"] - Copies the Pipfile, Pipfile.lock from host directory to container's current directory (which is /app)

2. We build the image from this Dockerfile to see the output. imagename is zoomcamp-test

>>> docker build -t zoomcamp-test .

```
$ ls
 Dockerfile   Pipfile.lock        ping.py   predict-test.py   train.py
 Pipfile      'model_C=1.0.bin'   plan.md   predict.py
$ docker build -t zoomcamp-test .
Sending build context to Docker daemon  33.28kB
Step 1/4 : FROM python:3.8.12-slim
 ---> 2e56f6b0af69
Step 2/4 : RUN pip install pipenv
 ---> Running in 080662b3bfa8
Removing intermediate container 080662b3bfa8
 ---> c0f436f873c8
Step 3/4 : WORKDIR /app
 ---> Running in 947745584c59
Removing intermediate container 947745584c59
 ---> 4a59e7ebf12f
Step 4/4 : COPY ["Pipfile", "Pipfile.lock", "./"]
 ---> 37a18209b54f
Successfully built 37a18209b54f
Successfully tagged zoomcamp-test:latest
```

3. We run this image

>>> docker run -it --rm entrypoint=bash zoomcamp-test

```
$ docker run -it --rm --entrypoint=bash zoomcamp-test
root@fbf742a0b8eb:/app# ls
Pipfile  Pipfile.lock
```

Now, see the Prompt is now within the /app directory of the container. And, it contains the Pipfile, Pipfile.lock. Here, we can manually run the command

>>> pipenv install → This will install all the dependencies listed in the Pipfile. But, we want to do this via the Dockerfile

Version 2

1. Adding more statements to previous Dockerfile

```
Dockerfile
1   FROM python:3.8.12-slim
2
3   RUN pip install pipenv
4
5   WORKDIR /app
6   COPY ["Pipfile", "Pipfile.lock", "./"]
7
8   RUN pipenv install --system --deploy
```

RUN pipenv install → This would create the Venv and install all the dependencies within it. But, since only the Churn Service will be there in this Docker Container and nothing else, we can skip creating the Virtual environment. For that, we use RUN pipenv install --system --deploy which will install the dependencies inside the Container's main Python folder

2. We build the image again

>>> docker build -t zoomcamp-test

```
$ docker build -t zoomcamp-test .
Sending build context to Docker daemon  33.28kB
Step 1/5 : FROM python:3.8.12-slim
 ---> 2e56f6b0af69
Step 2/5 : RUN pip install pipenv
 ---> Using cache
 ---> c0f436f973c8
Step 3/5 : WORKDIR /app
 ---> Using cache
 ---> 4a59e7ebf12f
Step 4/5 : COPY ["Pipfile", "Pipfile.lock", "./"]
 ---> Using cache
 ---> 37a18209b54f
Step 5/5 : RUN pipenv install --system --deploy
 ---> Running in d4c403e58e68
Installing dependencies from Pipfile.lock (ded843)...
```

All the previous steps are run from cache. New step is executed and it installs flask, gunicorn etc

3. Run this new image

>>> docker run -it --rm --entrypoint=bash zoomcamp-test

```
$ docker run -it --rm --entrypoint=bash zoomcamp-test
root@ee92f9c687fe:/app# ls
Pipfile  Pipfile.lock
root@ee92f9c687fe:/app# gunicorn

Error: No application module specified.
```

We run gunicorn without any arguments. It runs but it complains of no app module specified

Version 3-

1. Add statements to copy the model and predict.py file

```
Dockerfile
1   FROM python:3.8.12-slim
2
3   RUN pip install pipenv
4
5   WORKDIR /app
6   COPY ["Pipfile", "Pipfile.lock", "./"]
7
8   RUN pipenv install --system --deploy
9
0   COPY ["predict.py", "model_C=1.0.bin", "./"]
```

COPY ['predict.py', 'model.bin', './'] → Copy the Model and the Predict.py to the container's app folder

2. Re-Build the image

```
$ docker build -t zoomcamp-test .
Sending build context to Docker daemon  33.28kB
Step 1/6 : FROM python:3.8.12-slim
 ---> 2e56f6b0af69
Step 2/6 : RUN pip install pipenv
 ---> Using cache
 ---> c0f436f973c8
Step 3/6 : WORKDIR /app
 ---> Using cache
 ---> 4a59e7ebf12f
Step 4/6 : COPY ["Pipfile", "Pipfile.lock", "./"]
 ---> Using cache
 ---> 37a18209b54f
Step 5/6 : RUN pipenv install --system --deploy
 ---> Using cache
 ---> f5ccdd960de5
Step 6/6 : COPY ["predict.py", "model_C=1.0.bin", "./"]
```

3. Run the new image

```
$ docker run -it --rm --entrypoint=bash zoomcamp-test
root@5ab5e15d4d8d:/app# ls
 Pipfile   Pipfile.lock  'model_C=1.0.bin'   predict.py
```

The /app directory now has the model and predict.py file

4. Inside the container, you can run the webservice

container >>> gunicorn --bind=0.0.0.0:9696 predict:app

```
root@5ab5e15d4d8d:/app# gunicorn --bind=0.0.0.0:9696 predict:app
[2021-10-01 12:45:27 +0000] [9] [INFO] Starting gunicorn 20.1.0
[2021-10-01 12:45:27 +0000] [9] [INFO] Listening at: http://0.0.0.0:9696 (
9)
[2021-10-01 12:45:27 +0000] [9] [INFO] Using worker: sync
[2021-10-01 12:45:27 +0000] [11] [INFO] Booting worker with pid: 11
```

and it runs. But, we will not be able to talk to it since we have not mapped the host port and container port.

Version 5-

1. Expose the Container Port 9696 and set the default entrypoint of the container to run Gunicorn Predict:App on Port 9696

```
Dockerfile
1   FROM python:3.8.12-slim
2
3   RUN pip install pipenv
4
5   WORKDIR /app
6   COPY ["Pipfile", "Pipfile.lock", "./"]
7
8   RUN pipenv install --system --deploy
9
0   COPY ["predict.py", "model_C=1.0.bin", "./"]
1
2   EXPOSE 9696
3
4   ENTRYPOINT ["gunicorn", "--bind=0.0.0.0:9696", "predict:app"]
```

EXPOSE 9696 - Makes Container Port 9696 available for outside world communication
ENTRYPOINT [“”,””,””] - Put each word of the command ‘gunicorn --bind:0.0.0.0:9696 predict:app’ in double quotes and separate them using comma

2. Re-build the image

>>> docker build zoomcamp-test .

```
Step 6/8 : COPY ["predict.py", "model_C=1.0.bin", "./"]
 ---> Using cache
 ---> fac94a48c7f4
Step 7/8 : EXPOSE 9696
 ---> Running in dc915fa17157
Removing intermediate container dc915fa17157
 ---> 58f310638b29
Step 8/8 : ENTRYPOINT ["gunicorn", "--bind=0.0.0.0:9696", "predict:app"]
 ---> Running in 2c22ce57d65a
Removing intermediate container 2c22ce57d65a
 ---> 5b0677061d37
Successfully built 5b0677061d37
Successfully tagged zoomcamp-test:latest
```

3. Run the image specifying the host and container port → This will start the Predict Webservice

>>> docker run -it --rm -p 9696:9696 zoomcamp-test

```
$ docker run -it --rm -p 9696:9696 zoomcamp-test
[2021-10-01 12:52:11 +0000] [1] [INFO] Starting gunicorn 20.1.0
[2021-10-01 12:52:11 +0000] [1] [INFO] Listening at: http://0.0.0.0:9696 (
1)
[2021-10-01 12:52:11 +0000] [1] [INFO] Using worker: sync
[2021-10-01 12:52:11 +0000] [9] [INFO] Booting worker with pid: 9
```

-p HOST_PORT:CONTAINER_PORT
So, this command will directly run the Webservice using the gunicorn command as we configured in the Dockerfile

4. Run the Tester from another terminal and it communicates with the Webservice Docker Container

```
$ python predict-test.py
{'churn': False, 'churn_probability': 0.3257561103397851}
not sending promo email to xyz-123
```

Deploying the Docker image to Cloud (optional)

You can deploy the Docker image of the webservice on any Cloud provider like AWS, Azure, GCP, Heroku, Python anywhere

(Make notes later)
https://www.youtube.com/watch?v=HGPJ4ekhcLg&list=PL3MmuxUbc_hIhxl5Ji8t4O6lPAOpHaCLR&index=56

6: DECISION TREES & ENSEMBLE LEARNING: CREDIT RISK SCORING PROJECT

Imagine you want to buy a mobile phone, so you visit your bank to apply for a loan. You fill out an application form that requests various details, such as your income, the price of the phone, and the loan amount you need. The bank evaluates your application and assigns a score, ultimately deciding whether to approve or decline your request with a 'yes' or 'no' response.

In this project, our goal is to build a model that the bank can utilize to make informed decisions about lending money to customers. The bank can provide the model with customer information, and in return, the model will generate a risk score, indicating the likelihood of a customer defaulting on the loan. This risk score enables the bank to make well-informed lending decisions.

Our approach involves analyzing historical data from various customers and their loan applications. For each case, we have information about the requested loan amount and whether the customer successfully repaid the loan or defaulted.
For instance:

- Customer A –> OK
- Customer B –> OK
- Customer C –> DEFAULT
- Customer D –> DEFAULT
- Customer E –> OK

This problem can be framed as **binary classification**, where 'y' represents the target variable, and it can take on two values: 0 (OK) or 1 (DEFAULT). Our objective is to train a model to predict, for each new customer, the probability that they will default:

$g(x_i)$ –> PROBABILITY OF DEFAULT

We have 'X,' which encompasses all the customer information, and the target variable 'y,' which indicates the Probability of Default.

Dataset is at https://github.com/gastonstat/CreditScoring. The 'Status' variable in the dataset denotes whether the customer defaulted or not.

1 Status	credit status
2 Seniority	job seniority (years)
3 Home	type of home ownership
4 Time	time of requested loan
5 Age	client's age
6 Marital	marital status
7 Records	existence of records
8 Job	type of job
9 Expenses	amount of expenses
10 Income	amount of income
11 Assets	amount of assets
12 Debt	amount of debt
13 Amount	amount requested of loan

14 Price	price of good

In this project, we'll experiment with 3 models -

- Decision trees - Special algorithm that learns rules from the dataset like if-else rules
- Random forest - Many different Decision trees combined together creates a Random Forest
- Gradient boost/XGBoost - One way of combining multiple decision trees

And out of these 3 models, we'll select the best one

Data Cleanup & Preparation

We'll need to import several essential libraries:-

```
import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
```

Read the CSV file and Look at the first 5 rows

```
#df = pd.read_csv(data)
df = pd.read_csv('CreditScoring.csv')
df.head()
```

	Status	Seniority	Home	Time	Age	Marital	Records	Job	Expenses	Income	Assets	Debt	Amount	Price
0	1	9	1	60	30	2	1	3	73	129	0	0	800	846
1	1	17	1	60	58	3	1	1	48	131	0	0	1000	1658
2	2	10	2	36	46	2	2	3	90	200	3000	0	2000	2985
3	1	0	1	60	24	1	1	1	63	182	2500	0	900	1325
4	1	0	1	36	26	1	1	1	46	107	0	0	310	910

Make all column names in lowercase

```
df.columns = df.columns.str.lower()
df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	1	9	1	60	30	2	1	3	73	129	0	0	800	846
1	1	17	1	60	58	3	1	1	48	131	0	0	1000	1658
2	2	10	2	36	46	2	2	3	90	200	3000	0	2000	2985
3	1	0	1	60	24	1	1	1	63	182	2500	0	900	1325
4	1	0	1	36	26	1	1	1	46	107	0	0	310	910

Re-encode Categorical variables

Now, the categorical columns like status, marital, job, etc are having numerical values. Refer to the Preprocessing rules as implemented in R (https://github.com/gastonstat/CreditScoring/blob/master/Part1_CredScoring_Processing.R)

```
# change factor levels (i.e. categories)
levels(dd$Status) = c("good", "bad")
levels(dd$Home) = c("rent", "owner", "priv", "ignore", "parents", "other")
levels(dd$Marital) = c("single", "married", "widow", "separated", "divorced")
levels(dd$Records) = c("no_rec", "yes_rec")
levels(dd$Job) = c("fixed", "partime", "freelance", "others")
```

In R, the array values begin with 1. So, the values are like 1,2,3, etc. We'll re-encode the numerical values for these columns to text.

We check the value_counts for each value for each of these categorical variables

```
df.status.value_counts()
```
✓ 0.0s

```
status
1    3200
2    1254
0       1
```

```
df.home.value_counts()
```
✓ 0.0s

```
home
2    2107
1     973
5     783
6     319
3     247
4      20
0       6
```

```
df.marital.value_counts()
```
✓ 0.0s

```
marital
2    3241
1     978
4     130
3      67
5      38
0       1
```

```
df.records.value_counts()
```
✓ 0.0s

```
records
1    3682
2     773
```

```
df.job.value_counts()
```
✓ 0.0s

```
job
1    2806
3    1024
2     452
4     171
0       2
```

For Status, 1 → ok, 2 → default, 0→ unk

We can use .map() function to translate each numerical value to text. .map() function takes a dictionary as argument

```
status_values = {
    1: 'ok',
    2: 'default',
    0: 'unk'
}
```

,

```
df.status = df.status.map(status_values)
df.head()
```

	status	seniority	home
0	ok	9	1
1	ok	17	1
2	default	10	2
3	ok	0	1
4	ok	0	1

Similarly, we use .map() function to translate other categorical columns to text

```
home_values = {
    1: 'rent',
    2: 'owner',
    3: 'private',
    4: 'ignore',
    5: 'parents',
    6: 'other',
    0: 'unk'
}
df.home = df.home.map(home_values)

marital_values = {
    1: 'single',
    2: 'married',
    3: 'widow',
    4: 'separated',
    5: 'divorced',
    0: 'unk'
}
df.marital = df.marital.map(marital_values)
```

```
records_values = {
    1: 'no',
    2: 'yes',
    0: 'unk'
}
df.records = df.records.map(records_values)

job_values = {
    1: 'fixed',
    2: 'partime',
    3: 'freelance',
    4: 'others',
    0: 'unk'
}
df.job = df.job.map(job_values)

df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	income	Assets	debt	amount	price
0	ok	9	rent	60	30	married	no	freelance	73	129	0	0	800	846
1	ok	17	rent	60	58	widow	no	fixed	48	131	0	0	1000	1658
2	default	10	owner	36	46	married	yes	freelance	90	200	3000	0	2000	2985
3	ok	0	rent	60	24	single	no	fixed	63	182	2500	0	900	1325
4	ok	0	rent	36	26	single	no	fixed	46	107	0	0	310	910

Handle Missing values

Check for missing values in numerical columns. We get a quick summary using .describe() function

```
df.describe().round()
```

	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	763317.0	1060341.0	404382.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	8703625.0	10217569.0	6344253.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3500.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	166.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	99999999.0	99999999.0	99999999.0	5000.0	11140.0

Observe that the max values of Income, Assets & Debts are 99999999.0 which is too large and so that value is used as missing value indicator. We'll replace this string of 9's with NaN using replace(to_replace, value) function

```
df.income.max()
# Output: 99999999

df.income.replace(to_replace=99999999, value=np.nan)

df.income.replace(to_replace=99999999, value=np.nan).max()
# Output: 959.0
```

```
for c in ['income', 'assets', 'debt']:
    df[c] = df[c].replace(to_replace=99999999, value=np.nan)
```

	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4421.0	4408.0	4437.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	131.0	5403.0	343.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	86.0	11573.0	1246.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3000.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	165.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	959.0	300000.0	30000.0	5000.0	11140.0

Also, since we are working with Binary Classification for status column, it should contain only 2 values. So, for that one record with status = 'unk', we'll drop that record and reset the index

```
df = df[df.status != 'unk'].reset_index(drop=True)
df
```

	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	ok	9	rent	60	30	married	no	freelance	73	129.0	0.0	0.0	800	846
1	ok	17	rent	60	58	widow	no	fixed	48	131.0	0.0	0.0	1000	1658
2	default	10	owner	36	46	married	yes	freelance	90	200.0	3000.0	0.0	2000	2985
3	ok	0	rent	60	24	single	no	fixed	63	182.0	2500.0	0.0	900	1325
4	ok	0	rent	36	26	single	no	fixed	46	107.0	0.0	0.0	310	910
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4449	default	1	rent	60	39	married	no	fixed	69	92.0	0.0	0.0	900	1020
4450	ok	22	owner	60	46	married	no	fixed	60	75.0	3000.0	600.0	950	1263
4451	default	0	owner	24	37	married	no	partime	60	90.0	3500.0	0.0	500	963
4452	ok	0	rent	48	23	single	no	freelance	49	140.0	0.0	0.0	550	550
4453	ok	5	owner	60	32	married	no	freelance	60	140.0	4000.0	1000.0	1350	1650

Train-Val-Test Split

```
from sklearn.model_selection import train_test_split

df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=11
df_train, df_val = train_test_split(df_full_train, test_size=0.25, random_st

df_train = df_train.reset_index(drop=True)
df_val = df_val.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)
```

We split the dataset into Train (60%), Validation (20%), Test (20%). Full Train = Train + Validation (80%). Also, we reset the indexes

Now, status is our target variable which needs to be 0 or 1. So, we convert it back to 0 & 1 using astype() [default → 1, ok → 0]

```
df_train.status
# Output:
# 0        default
# 1        default
# 2             ok
# 3        default
# 4             ok
#             ...
# 2667          ok
# 2668          ok
# 2669          ok
# 2670          ok
# 2671          ok
```

```
(df_train.status == 'default').astype('int')

# Output:
# 0       1
# 1       1
# 2       0
# 3       1
# 4       0
#         ..
# 2667    0
# 2668    0
# 2669    0
# 2670    0
# 2671    0
```

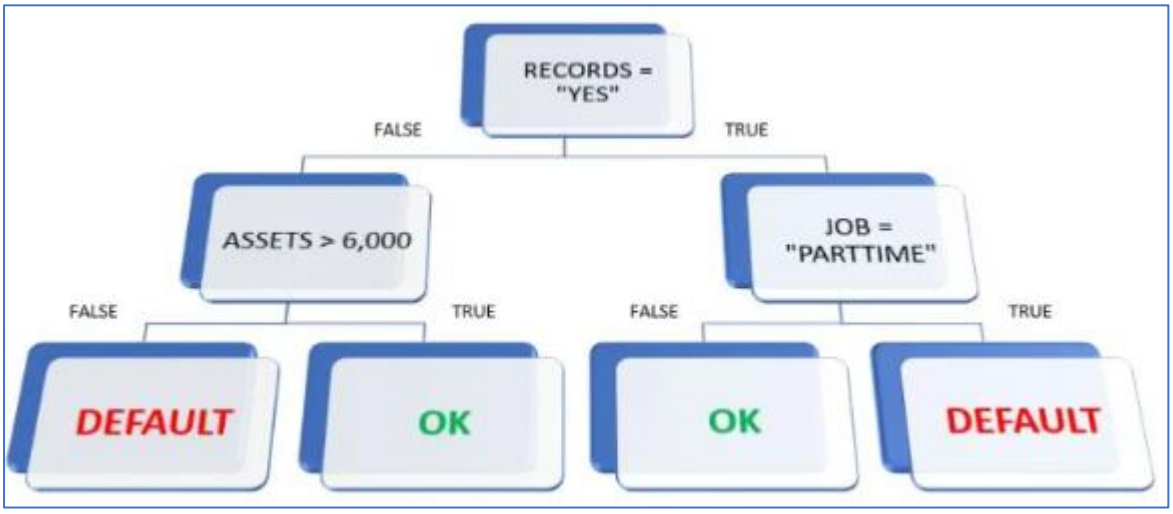
```
y_train = (df_train.status == 'default').astype('int').values
y_val = (df_val.status == 'default').astype('int').values
y_test = (df_test.status == 'default').astype('int').values

del df_train['status']
del df_val['status']
del df_test['status']

df_train
```

Create target vectors, y_train, y_val, y_test from respective df.values. Delete the Status column from each Split set to avoid its accidental usage in Model Training

	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	10	owner	36	36	married	no	freelance	75	0.0	10000.0	0.0	1000	1400
1	6	parents	48	32	single	yes	fixed	35	85.0	0.0	0.0	1100	1330
2	1	parents	48	40	married	no	fixed	75	121.0	0.0	0.0	1320	1600
3	1	parents	48	23	single	no	partime	35	72.0	0.0	0.0	1078	1079
4	5	owner	36	46	married	no	freelance	60	100.0	4000.0	0.0	1100	1897
...	...	...	...	...	...	...	...	...	...	...	...	...	...
2667	18	private	36	45	married	no	fixed	45	220.0	20000.0	0.0	800	1600
2668	7	private	60	29	married	no	fixed	60	51.0	3500.0	500.0	1000	1290
2669	1	parents	24	19	single	no	fixed	35	28.0	0.0	0.0	400	600
2670	15	owner	48	43	married	no	freelance	60	100.0	18000.0	0.0	2500	2976
2671	12	owner	48	27	married	yes	fixed	45	110.0	5000.0	1300.0	450	1636

ABOUT DECISION TREES

Decision trees are a tree-like structure where each internal node represents a feature or attribute, each branch signifies a decision or outcome, and each leaf node provides a final prediction or classification. From each node, there is one arrow to the left (condition = false) and one to the right (condition=true). Then there is the next condition which can be true or false. … until there is the final decision 'OK' or 'DEFAULT'.



So, assume our decision tree is as above. If "RECORDS = YES" is true, then check for "JOB=PARTTIME" condition - if true, then decision is DEFAULT; if false, then decision is OK. If "RECORDS = YES" is false, then check for "ASSETS > 6000" condition - if true, then decision is OK; if false, then decision is DEFAULT. We can implement the above rules in code like below and test with one sample record-

```
def assess_risk(client):
    if client['records'] == 'yes':
        if client['job'] == 'parttime':
            return 'default'
        else:
            return 'ok'
    else:
        if client['assets'] > 6000:
            return 'ok'
        else:
            return 'default'
```

```
# just take one record to test
xi = df_train.iloc[0].to_dict()
xi
# Output:
# {'seniority': 10,
# 'home': 'owner',
# 'time': 36,
# 'age': 36,
# 'marital': 'married',
# 'records': 'no',
# 'job': 'freelance',
# 'expenses': 75,
# 'income': 0.0,
# 'assets': 10000.0,
# 'debt': 0.0,
# 'amount': 1000,
# 'price': 1400}
```

```
assess_risk(xi)
# Output: 'ok'
```

For the sample record, first condition is "RECORDS = YES." Our client has no records, so we go to the left. The second condition is "ASSETS > 6000." The assets of our client are 10,000, so we go to the right. Now we reach the decision node, which in this case is "OK."

Now, we do not want to hard-code these rules but want the model to identify and learn the rules from the dataset itself. This is the difference between DECISION TREES and RULE-BASED system.

Note- DECISION TREES can be used for both Classification (DecisionTreeClassifier) & Regression (DecisionTreeRegressor)

Using SKlearn-Decision Tree

from sklearn.tree import DecisionTreeClassifier

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction import DictVectorizer
from sklearn.metrics import roc_auc_score
```

We first need to import necessary packages. From Scikit-Learn, we import DecisionTreeClassifier. Because we have categorical variables, we also need to import DictVectorizer as seen before. Also, we will import roc_auc_curve from sklearn.metrics to compute the Area Under Curve to evaluate our model's performance

Now we need to turn our training dataframe df_train into a list of dictionaries and then turn this list of dictionaries into the feature matrix. Since our df_train has NaN values, we replace them with 0 using fillna(0).

```
# This will lead us later to an error
#train_dicts = df_train.to_dict(orient='records')
# ... so we use instead:

train_dicts = df_train.fillna(0).to_dict(orient='records')
train_dicts[:5]
```

```
# Output:
#[{'seniority': 10,
#  'home': 'owner',
#  'time': 36,
#  'age': 36,
#  'marital': 'married',
#  'records': 'no',
#  'job': 'freelance',
#  'expenses': 75,
#  'income': 0.0,
#  'assets': 10000.0,
#  'debt': 0.0,
#  'amount': 1000,
#  'price': 1400},
# {'seniority': 6,
```

After that, we use DictVectorizer to do One-hot encoding of Categorical variables and then prepare the X_train matrix by dv.fit_transform(train_dicts)

```
dv = DictVectorizer(sparse=False)
X_train = dv.fit_transform(train_dicts)
X_train

# Output:
# array([[3.60e+01, 1.00e+03, 1.00e+04, ...,
#             [3.20e+01, 1.10e+03, 0.00e+00,
#             [4.00e+01, 1.32e+03, 0.00e+00,
#             ...,
#             [1.90e+01, 4.00e+02, 0.00e+00,
#             [4.30e+01, 2.50e+03, 1.80e+04,
#             [2.70e+01, 4.50e+02, 5.00e+03,
```

All the numerical features remain unchanged, but we have encoding for categorical features.

```
dv.get_feature_names_out()

# Ouput:
# array(['age', 'amount', 'assets', 'debt', 'expenses', 'home=ignore',
#             'home=other', 'home=owner', 'home=parents', 'home=private',
#             'home=rent', 'home=unk', 'income', 'job=fixed', 'job=freelance',
#             'job=others', 'job=partime', 'job=unk', 'marital=divorced',
#             'marital=married', 'marital=separated', 'marital=single',
#             'marital=unk', 'marital=widow', 'price', 'records=no',
#             'records=yes', 'seniority', 'time'], dtype=object)
```

Next,

- we create object dt of DecisionTreeClassifier() class.
- We train the DecisionTree using dt.fit(x_train, y_train).
- We convert the Validation dataframe, df_val to dictionary.
- Then, extract the Validation Feature Matrix X_val using dv.transform(val_dicts).
- Then, we use the trained DecisionTree to make predictions, y_pred on the Validation dataset X_val.
- And compute the Area Under Curve by comparing y_pred, y_val.

```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)

val_dicts = df_val.fillna(0).to_dict(orient='records')
X_val = dv.transform(val_dicts)
y_pred = dt.predict_proba(X_val)[:, 1]

roc_auc_score(y_val, y_pred)
# Output: 0.6547098641350415
```

AUC = 0.65 (0.5 < AUC < 1) which is close to that of random model and is not Good.

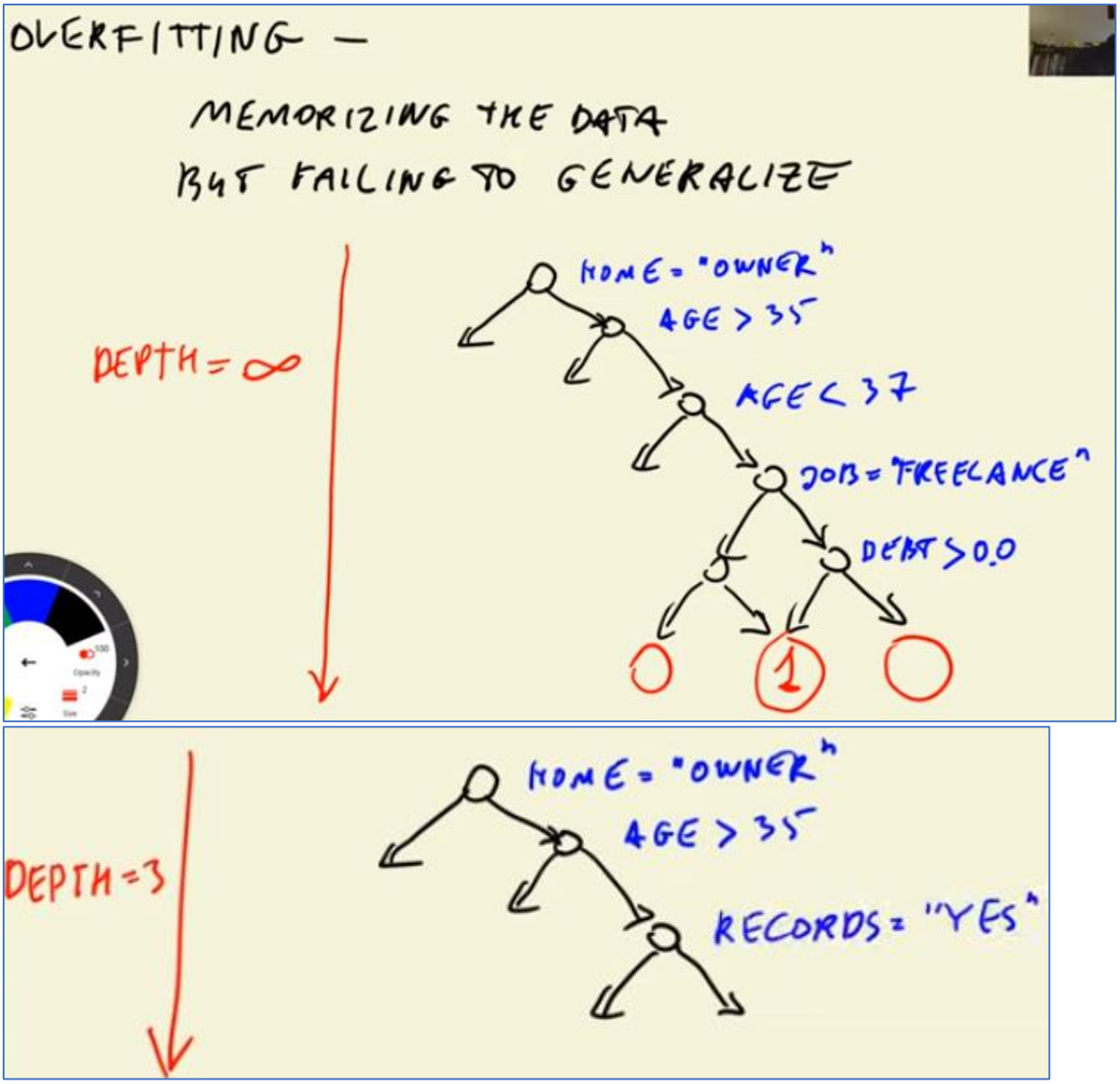
Overfitting leading to poor performance

Let us understand why the AUC till now is not good and what we can do to improve it. Let us check the AUC by first predicting using the Training dataset and then compare the predictions with the actuals

```
y_pred = dt.predict_proba(X_train)[:, 1]
roc_auc_score(y_train, y_pred)
# Output: 1.0
```

See that the AUC = 1 which means that if you provide example from training data, the model's predictions are correct for every sample → The model has fully memorized the training data and so if we give it an unseen sample, it is clueless because it is trying to "fit" the unseen sample with already learnt sample and it finds it difficult to find a close match → This is called **OVERFITTING** as it memorizes the whole data to a deep level and is not able to generalize it for other unseen variations.

Overfitting happens when we let the decision tree create specific rules for more nested conditions leading to a deeper tree. The deeper the tree, the higher the Overfitting and the more exact match it tries to find which can lead to poor performance for unseen examples



So, if we restrict the tree to only grow up to three levels deep, the tree will learn rules that are less specific and would be able to perform better with unseen samples

We can set the depth when creating "dt" through max_depth argument. Here we set max_depth=3 and re-train the model.

```
dt = DecisionTreeClassifier(max_depth=3)
dt.fit(X_train, y_train)

y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train', auc)
# Output: train 0.7761016984958594

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('val', auc)
# Output: val 0.7389079944782155
```

And we see the AUC of Predictions on Training data = 0.77 (dropped from 1.0 so overfitting reduced) and AUC of Predictions on Validation data = 0.74 (improved from 0.65)

Decision stump - A decision tree with just one level (i.e. max depth = 1 or only one condition). Let us see the performance for Decision stump in our case


```
dt = DecisionTreeClassifier(max_depth=1)
dt.fit(X_train, y_train)

y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train', auc)
# Output: train 0.6282660131823559

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('val', auc)
# Output: val 0.6058644740984719
```

And we see the AUC of Predictions on Training data = 0.62 and AUC of Predictions on Validation data = 0.60 which is even worse than Overfitted version

Visualizing the Decision Tree

Visualizing the tree means to inspect what rules the Decision tree has learnt. For this, we use sklearn.tree import export_text

For max_depth = 1 (Decision Stump)

```
from sklearn.tree import export_text
print(export_text(dt))

# Output:
# |--- feature_25 <= 0.50
# |   |--- class: 1
# |--- feature_25 >  0.50
# |   |--- class: 0
```

```
# in the video Alexey uses
# print(export_text(dt,feature_names=dv.get_feature_names()))

names = dv.get_feature_names_out().tolist()
print(export_text(dt, feature_names=names))

# Output:
# |--- records=no <= 0.50
# |   |--- class: 1
# |--- records=no >  0.50
# |   |--- class: 0
```

The first output shows “feature_25” so to know which feature exactly is “feature_25”, we use dv.get_feature_names with export_text(dt)

So, “records=no <= 0.5” means if “records=no” is FALSE, then its decision is 1 (DEFAULT) else it is 0 (OK) → This is the Rule the DT learnt

For max_depth = 2 (Two-level depth)

```
dt = DecisionTreeClassifier(max_depth=2)
dt.fit(X_train, y_train)

y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train', auc)
# Output: train 0.7054989859726213

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('val', auc)
# Output: val 0.6685264343319367
```

AUC of Predictions on Training data = 0.70 and AUC of Predictions on Validation data = 0.66 which is better than Overfitted version

```
print(export_text(dt))

# Output:
# |--- feature_26 <= 0.50
# |    |--- feature_16 <= 0.50
# |    |    |--- class: 0
# |    |--- feature_16 >  0.50
# |    |    |--- class: 1
# |--- feature_26 >  0.50
# |    |--- feature_27 <= 6.50
# |    |    |--- class: 1
# |    |--- feature_27 >  6.50
# |    |    |--- class: 0
```

```
names = dv.get_feature_names_out().tolist()
print(export_text(dt, feature_names=names))

# Output:
# |--- records=yes <= 0.50
# |    |--- job=partime <= 0.50
# |    |    |--- class: 0
# |    |--- job=partime >  0.50
# |    |    |--- class: 1
# |--- records=yes >  0.50
# |    |--- seniority <= 6.50
# |    |    |--- class: 1
# |    |--- seniority >  6.50
# |    |    |--- class: 0
```

So, here the Decision rules are as follows-

If “records = yes” is False

If “job=Parttime” is False

then “OK”

Else if “job=Parttime” is True

then “DEFAULT”

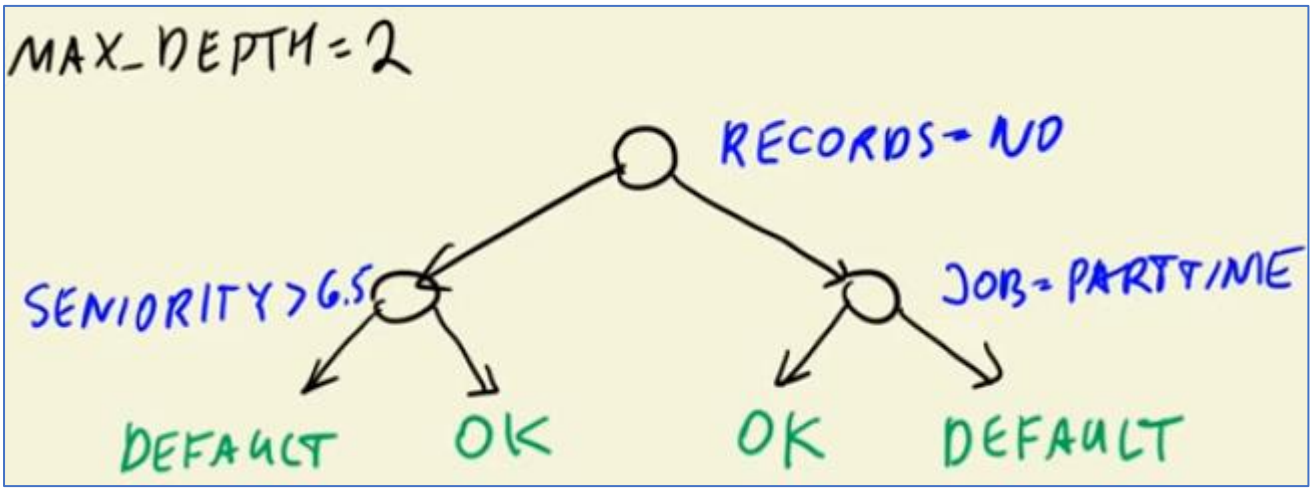
If “records=yes” is True

If “seniority <= 6.5”

then “DEFAULT”

else if “seniority > 6.5”

then “OK”



How Decision tree Learning algorithm works?

Lets understand some terms associated with Decision Tree by using a single level tree called Decision Stump.

- At that level, we have a **CONDITION NODE** which are of the form **FEATURE > T** where T is a threshold. This condition “feature > T” is called a **SPLIT**.
- This split divides the data into records that satisfy the condition (TRUE) and those that do not (FALSE).
- At the top of the tree, we have the condition node, and at the bottom, there are the **LEAVES** of the tree. These leaves are called **DECISION NODES**. Decision nodes are where the tree doesn’t go any deeper, and this is where decisions are made.
- In this context, “decision making” means that we choose the most frequently occurring status label within each group and use it as the final decision for that group.

FEATURE > T

/ \

/ FALSE \ TRUE

/ \
DEFAULT OK

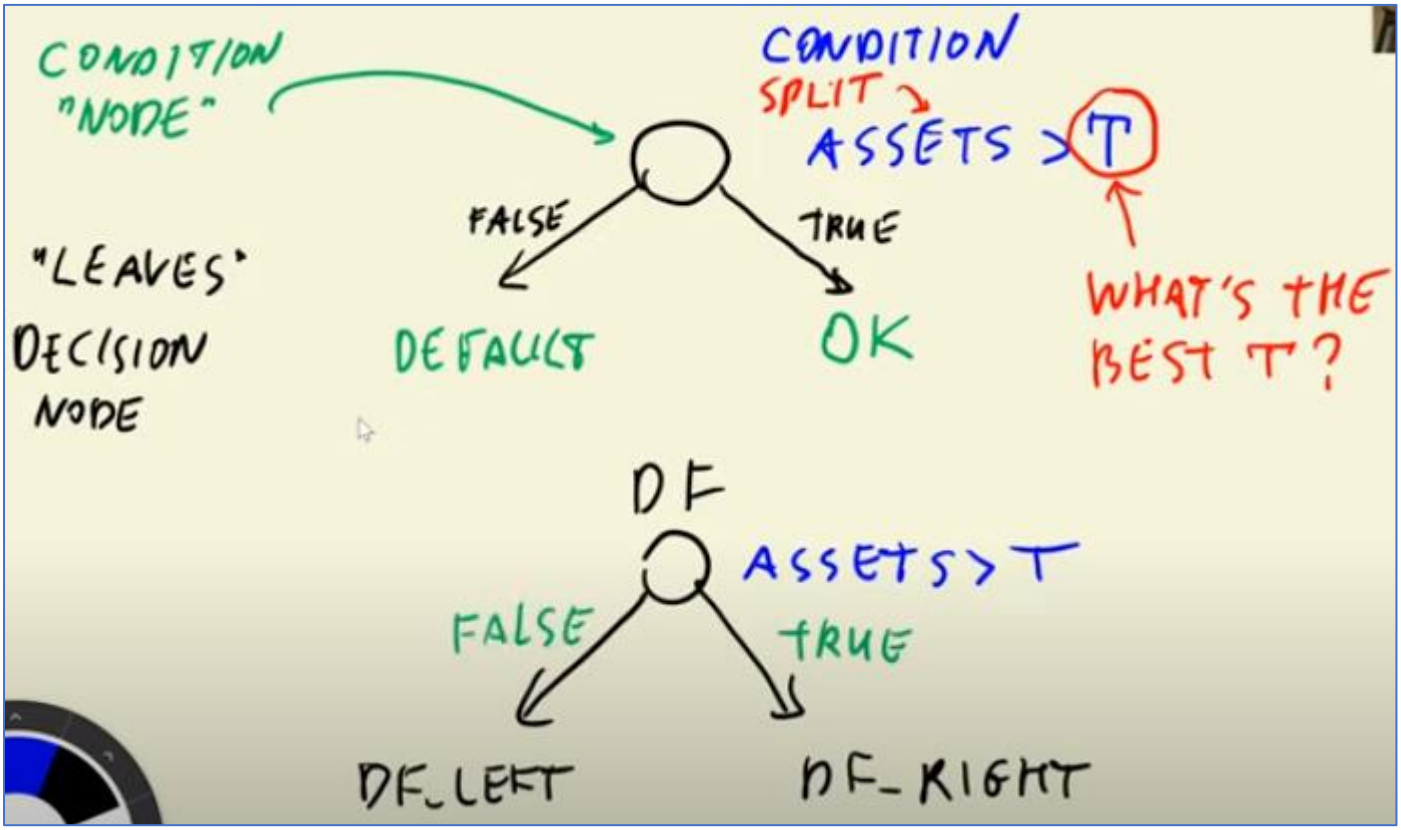
Simple one-feature dataset

To illustrate how the learning algorithm works, let's use a simple dataset. This dataset contains 8 records with one value for 'assets' and one for 'status,' which is our target variable.

```
data = [
    [8000, 'default'],
    [2000, 'default'],
    [   0, 'default'],
    [5000, 'ok'],
    [5000, 'ok'],
    [4000, 'ok'],
    [9000, 'ok'],
    [3000, 'default'],
]

df_example = pd.DataFrame(data, columns=['assets', 'status'])
df_example
```

	assets	Status
0	8000	Default
1	2000	default
2	0	default
3	5000	ok
4	5000	ok
5	4000	ok
6	9000	ok
7	3000	default

We want to train a decision tree using this column "ASSETS" i.e. the condition will be "ASSETS > T". We want to split the dataset into two parts. One part where the condition holds true, and the other part where the condition does not hold true. And, for the condition "ASSETS > T", we want to know what is best value for T?



DF_LEFT is where this condition is false, so for all records where 'assets' is not greater than T.
DF_RIGHT is where this condition is true, so for all records where 'assets' is greater than T.
This is called **splitting**, so we're splitting the data set into two parts where it's true and where it's false.

To understand the best value of T, first lets sort the dataframe df_example by Assets in ascending order

```
df_example.sort_values('assets')
```

	Assets	Status
2	0	default
1	2000	default

	Assets	Status
7	3000	default
5	4000	ok
3	5000	ok
4	5000	ok
0	8000	default
6	9000	Ok

So, the thresholds T can be 0, 2000, 3000, 4000, 5000, 8000. We don't take 9000 as threshold because ASSETS > 9000 will have no records (no DF_RIGHT)
The possibilities for the threshold "T" are:

- T = 0 (ASSETS<=0 becomes LEFT, ASSETS>0 becomes RIGHT)
- T = 2000 (ASSETS<=2,000 becomes LEFT, ASSETS>2,000 becomes RIGHT)
- T = 3000 (ASSETS<=3,000 becomes LEFT, ASSETS>3,000 becomes RIGHT)
- T = 4000 (ASSETS<=4,000 becomes LEFT, ASSETS>4,000 becomes RIGHT)
- T = 5000 (ASSETS<=5,000 becomes LEFT, ASSETS>5,000 becomes RIGHT)
- T = 8000 (ASSETS<=8,000 becomes LEFT, ASSETS>8,000 becomes RIGHT)

So, for each threshold, we split the df_example into df_left and df_right and use some metric to determine which is the best threshold

```
Ts = [0, 2000, 3000, 4000, 5000, 8000]
```

```
from IPython.display import display

for T in Ts:
    print(T)
    df_left = df_example[df_example.assets <= T]
    df_right = df_example[df_example.assets > T]

    display(df_left)
    display(df_right)

    print()
```

Since we want to display two dataframes inside a for loop, Jupyter notebook will not print them by default. So, we need to use display() function from IPython.display

T	DF_LEFT	DF_RIGHT
T = 0	**Assets** **status** **2** 0 default	**Assets** **Status** **0** 8000 default **1** 2000 default **3** 5000 ok **4** 5000 ok **5** 4000 ok **6** 9000 ok **7** 3000 default
T = 2000	**Assets** **status** 1 2000 default 2 0 default	**Assets** **Status** **0** 8000 default **3** 5000 ok **4** 5000 ok **5** 4000 ok **6** 9000 ok

		7 3000 default
T = 3000		
	Assets **status** 1 2000 default 2 0 Default 7 3000 default	**Assets** **Status** **0** 8000 default **3** 5000 ok **4** 5000 ok **5** 4000 ok **6** 9000 ok
T = 4000		
	Assets **status** **1** 2000 default **2** 0 Default **5** 4000 ok **7** 3000 default	**Assets** **Status** **0** 8000 default **3** 5000 ok **4** 5000 ok **6** 9000 Ok
T = 5000		
	Assets **status** **1** 2000 default **2** 0 Default **3** 5000 ok **4** 5000 ok **5** 4000 ok **7** 3000 default	**Assets** **Status** **0** 8000 default **6** 9000 Ok
T = 8000		
	Assets **status** **0** 8000 default **1** 2000 default **2** 0 Default **3** 5000 ok **4** 5000 ok **5** 4000 ok **7** 3000 default	**Assets** **Status** **6** 9000 Ok

Now that we have generated many splits using different threshold values, we need to determine which split is the best. To evaluate this, we can use various split evaluation criteria. Let's take the example where T=4000. In this case, on the LEFT side, we have 3 cases labeled as DEFAULT and 1 case labeled as OK. On the RIGHT side, we have 3 cases labeled as OK and 1 case labeled as DEFAULT. As mentioned earlier, we select the most frequently occurring status within each group and use it as the final decision. In the case of the LEFT group, the most frequent status is "default," and for the RIGHT group, it's "ok."

```
T = 4000
df_left = df_example[df_example.assets <= T]
df_right = df_example[df_example.assets > T]

display(df_left)
display(df_right)
```

	Assets	status
1	2000	default
2	0	Default
5	4000	ok
7	3000	default

	Assets	Status
0	8000	default
3	5000	ok
4	5000	ok
6	9000	Ok

Misclassification Rate

There are many advanced metrics like Gini, Entropy that are actually used by the Scikit Learn algorithm. These metrics are collectively called **IMPURITY**. But, for simplicity of understanding, we will use **MISCLASSIFICATION RATE** as metric. **MISCLASSIFICATION RATE** measures the fraction of errors where the outcome does not match the Most Frequent Outcome of that group.

For T = 4000

	Assets	status
1	2000	default
2	0	Default
5	4000	ok
7	3000	default

DF_LEFT

	Assets	Status
0	8000	default
3	5000	ok
4	5000	ok
6	9000	Ok

DF_RIGHT

For T = 4000, for the DF_LEFT Group, the Most frequent outcome is DEFAULT. So, there are 3 correct and 1 wrong. So, Misclassification Rate = ¼ = 25%.
Similarly, for the DF_RIGHT Group, the Most frequent outcome is OK. So, there are 3 correct and 1 wrong. So, Misclassification Rate = ¼ = 25%.
So, the Misclassification Rate for T = 4000 is the average of its Left & Right Misclassification Rate = (25+25)/2 = 25%.
(To be more precise, we should take the Weighted Average but to keep it simple to understand, lets take simple average)

We can use the % of not matching outcome (.status.value_counts(normalize=True)) as the Misclassification rate.

```
T = 4000
df_left = df_example[df_example.assets <= T]
df_right = df_example[df_example.assets > T]

display(df_left)
print(df_left.status.value_counts(normalize=True))
display(df_right)
print(df_left.status.value_counts(normalize=True))
```

```
default    0.75
ok         0.25
```

,

```
ok         0.75
default    0.25
```

For Each T, we compute the Impurity of Left & Right

T	DF_LEFT with Misclassification Ratio	DF_RIGHT with Misclassification Ratio
T = 0	**Assets** **status** **2** 0 default default 1.0 Impurity Left = 0%	**Assets** **Status** **0** 8000 default **1** 2000 default **3** 5000 ok **4** 5000 ok **5** 4000 ok **6** 9000 ok **7** 3000 Default ok 0.571429 default 0.428571 Impurity Right = 43%
T = 2000	**Assets** **Status** 1 2000 default 2 0 default	**Assets** **Status** **0** 8000 default **3** 5000 ok **4** 5000 ok **5** 4000 ok

<table>
<tr><td></td><td>default 1.0
Impurity Left = 0%</td><td>6 9000 ok
7 3000 default

ok 0.666667
default 0.333333
Impurity Right = 33%</td></tr>
<tr><td>T = 3000</td><td>Assets status
1 2000 default
2 0 Default
7 3000 default

default 1.0
Impurity Left = 0%</td><td>Assets Status
0 8000 default
3 5000 ok
4 5000 ok
5 4000 ok
6 9000 ok

ok 0.8
default 0.2
Impurity Right = 20%</td></tr>
<tr><td>T = 4000</td><td>Assets status
1 2000 default
2 0 Default
5 4000 ok
7 3000 default
default 0.75
ok 0.25
Impurity Left = 25%</td><td>Assets Status
0 8000 default
3 5000 ok
4 5000 ok
6 9000 Ok
ok 0.75
default 0.25
Impurity Right = 25%</td></tr>
<tr><td>T = 5000</td><td>Assets Status
1 2000 default
2 0 Default
3 5000 ok
4 5000 ok
5 4000 ok
7 3000 default

default 0.5
ok 0.5
Impurity Left = 50%</td><td>Assets Status
0 8000 default
6 9000 Ok

default 0.5
ok 0.5
Impurity Right = 50%</td></tr>
<tr><td>T = 8000</td><td>Assets status
0 8000 default
1 2000 default
2 0 Default
3 5000 ok
4 5000 ok
5 4000 ok
7 3000 default

default 0.571429
ok 0.428571
Impurity Left = 43%</td><td>Assets Status
6 9000 Ok

ok 1.0
Impurity Right = 0%</td></tr>
</table>

Impurity Concept

The misclassification rate is just one way of measuring **impurity**. We aim for our leaves to be as pure as possible, meaning that the groups resulting from the split should ideally contain only observations of one class, making them ***pure***. The misclassification rate informs us of how impure these groups are by quantifying the error rate in classifying observations.

There are also alternative methods to measure impurity. Scikit-Learn offers more advanced criteria for splitting, such as entropy and Gini impurity. These criteria provide different ways to evaluate the quality of splits, with the goal of creating decision trees with more homogeneous leaves.

Now, back to the example, we tabulate the Impurity Rate as below-

T	Decision LEFT	Impurity LEFT	Decision RIGHT	Impurity RIGHT	AVG
0	DEFAULT	0%	OK	43%	21%
2000	DEFAULT	0%	OK	33%	16%
3000	**DEFAULT**	**0%**	**OK**	**20%**	**10%**
4000	DEFAULT	25%	OK	25%	25%
5000	DEFAULT	50%	OK	50%	50%
8000	DEFAULT	43%	OK	0%	21%

From the above table, we see that the split with T = 3000 gives lowest average Misclassification Rate (10%) → So, the condition is ASSETS > 3000

This is how we determine the best split when dealing with just one column. Summary of approach-

- We sort the dataset.
- We identify all possible thresholds.
- For each of the thresholds, we split the dataset.
- For each split, we calculate the impurity on the left and the right.
- We then compute the average impurity.
- Among all the splits considered, we select the one with the lowest average impurity.

Simple Two-Feature Dataset

Now, we will explore how to find the best split when we have two feature columns. This will lead us to a more general algorithm for finding the best split when working with multiple features. To the previous 1-feature dataset, lets add “debt” feature

```
data = [
    [8000, 3000, 'default'],
    [2000, 1000, 'default'],
    [   0, 1000, 'default'],
    [5000, 1000, 'ok'],
    [5000, 1000, 'ok'],
    [4000, 1000, 'ok'],
    [9000,  500, 'ok'],
    [3000, 2000, 'default'],
]

df_example = pd.DataFrame(data, columns=['assets', 'debt', 'status'])
df_example
```

	ASSETS	Debt	Status
0	8000	3000	default
1	2000	1000	default
2	0	1000	Default
3	5000	1000	Ok
4	5000	1000	Ok
5	4000	1000	Ok
6	9000	500	Ok
7	3000	2000	Default

In the previous section, we already saw the split using “Assets” feature. Now, let’s investigate the results of attempting a split based on the ‘debt’ feature. To do this, we should first sort the data by the ‘debt’ column.

```
df_example.sort_values('debt')
```

	ASSETS	Debt	Status
6	9000	500	ok
1	2000	1000	default
2	0	1000	default
3	5000	1000	ok
4	5000	1000	ok
5	4000	1000	ok
7	3000	2000	default
0	8000	3000	default

Possible thresholds:

- T = 500 (DEBT<=500 becomes LEFT, DEBT>500 becomes RIGHT)
- T = 1000 (DEBT<=1,000 becomes LEFT, DEBT>1,000 becomes RIGHT)
- T = 2000 (DEBT<=2,000 becomes LEFT, DEBT>2,000 becomes RIGHT)

So we’ve seen how it works for only one feature. Let’s generalize a bit and put both threshold variables in a dictionary. If there are more features, we can put them here as well.

```
thresholds = {
    'assets': [0, 2000, 3000, 4000, 5000, 8000],
    'debt': [500, 1000, 2000]
}

for feature, Ts in thresholds.items():
    print('####################')
    print(feature)
    for T in Ts:
        print(T)
        df_left = df_example[df_example[feature] <= T]
        df_right = df_example[df_example[feature] > T]

        display(df_left)
        print(df_left.status.value_counts(normalize=True))
        display(df_right)
        print(df_right.status.value_counts(normalize=True))

        print()
    print('####################')
```

For Assets feature

T	DF_LEFT with Misclassification Ratio				DF_RIGHT with Misclassification Ratio			
T = 0		Assets	Debt	status		Assets	Debt	Status
	2	0	1000	default	**0**	8000	3000	default

	default 1.0 Impurity Left = 0%	1 2000 1000 default 3 5000 1000 ok 4 5000 1000 ok 5 4000 1000 ok 6 9000 500 ok 7 3000 2000 Default ok 0.571429 default 0.428571 Impurity Right = 43%
T = 2000	**Assets Debt Status** 1 2000 1000 default 2 0 1000 default default 1.0 Impurity Left = 0%	**Assets Debt Status** 0 8000 3000 default 3 5000 1000 ok 4 5000 1000 ok 5 4000 1000 ok 6 9000 500 ok 7 3000 2000 default ok 0.666667 default 0.333333 Impurity Right = 33%
T = 3000	**Assets Debt status** 1 2000 1000 default 2 0 1000 Default 7 3000 2000 default default 1.0 Impurity Left = 0%	**Assets Debt Status** 0 8000 3000 default 3 5000 1000 ok 4 5000 1000 ok 5 4000 1000 ok 6 9000 2000 ok ok 0.8 default 0.2 Impurity Right = 20%
T = 4000	**Assets Debt status** 1 2000 1000 default 2 0 1000 Default 5 4000 1000 ok 7 3000 2000 default default 0.75 ok 0.25 Impurity Left = 25%	**Assets Debt Status** 0 8000 3000 default 3 5000 1000 ok 4 5000 1000 ok 6 9000 2000 Ok ok 0.75 default 0.25 Impurity Right = 25%
T = 5000	**Assets Debt Status** 1 2000 1000 default 2 0 1000 Default 3 5000 1000 ok 4 5000 1000 ok 5 4000 1000 ok 7 3000 2000 default	**Assets Debt Status** 0 8000 3000 default 6 9000 2000 Ok default 0.5 ok 0.5 Impurity Right = 50%

	default 0.5 ok 0.5 Impurity Left = 50%	
T = 8000	**Assets Debt status** **0** 8000 3000 default **1** 2000 1000 default **2** 0 1000 Default **3** 5000 1000 ok **4** 5000 1000 ok **5** 4000 1000 ok **7** 3000 2000 default default 0.571429 ok 0.428571 **Impurity Left = 43%**	**Assets Debt Status** **6** 9000 2000 Ok ok 1.0 Impurity Right = 0%

For Debt feature

T	DF_LEFT with Misclassification Ratio	DF_RIGHT with Misclassification Ratio
T = 500	**Assets Debt status** **6** 9000 500 ok default 1.0 Impurity Left = 0%	**Assets Debt Status** **0** 8000 3000 default **1** 2000 1000 default **2** 0 1000 default **3** 5000 1000 ok **4** 5000 1000 ok **5** 4000 1000 ok **7** 3000 2000 Default default 0.571429 ok 0.428571 Impurity Right = 43%
T = 1000	**Assets Debt Status** 1 2000 1000 default 2 0 1000 Default **3** 5000 1000 ok **4** 5000 1000 ok **5** 4000 1000 ok **6** 9000 500 ok ok 0.666667 default 0.333333 Impurity Left = 33%	**Assets Debt Status** **0** 8000 3000 default **7** 3000 2000 default default 1.0 Impurity Right = 0%
T = 2000	**Assets Debt Status** 1 2000 1000 default 2 0 1000 Default **3** 5000 1000 ok **4** 5000 1000 ok **5** 4000 1000 ok	**Assets Debt Status** **0** 8000 3000 default default 1.0 Impurity Right = 0%

	6 9000 500 Ok **7** 3000 2000 default ok 0.571429 default 0.428571 Impurity Left = 43%	

Impurity Table for Assets-

T	Decision LEFT	Impurity LEFT	Decision RIGHT	Impurity RIGHT	AVG
0	DEFAULT	0%	OK	43%	21%
2000	DEFAULT	0%	OK	33%	16%
3000	**DEFAULT**	**0%**	**OK**	**20%**	**10%**
4000	DEFAULT	25%	OK	25%	25%
5000	DEFAULT	50%	OK	50%	50%
8000	DEFAULT	43%	OK	0%	21%

Impurity Table for Debt-

T	Decision LEFT	Impurity LEFT	Decision RIGHT	Impurity RIGHT	AVG
500	OK	0%	DEFAULT	43%	21%
1000	OK	33%	DEFAULT	0%	16%
2000	OK	43%	DEFAULT	0%	21%

Now, impurity for Asset > 3000 is lower than any condition for Debt. So, the best split overall remains ASSET > 3000. If we were working with three features, we would have another group in the table with another set of rows and another variable to consider.

BEST SPLIT for Multiple features

```
FOR each feature in FEATURES:
    FIND all thresholds for the feature
    FOR each threshold in thresholds:
        SPLIT the dataset using "feature > threshold" condition
        COMPUTE the impurity of this split
SELECT the condition with the LOWEST IMPURITY
```

Since we apply the above recursive splitting, there has to be some Stopping Criteria to determine when to stop

STOPPING CRITERIA

To determine when to halt the recursive splitting process, it is essential to define stopping criteria. These criteria are vital for preventing overfitting and ensuring that the tree-building process concludes when specific conditions are satisfied.

The most common stopping criteria are:

1. **The group is already pure**:
   If a node is pure, meaning all samples in that node belong to the same class, further splitting is unnecessary as it won't provide any additional information.
2. **The tree reached depth limit (controlled by max_depth)**:
   Limiting the depth of the tree helps control overfitting and ensures that the tree doesn't become too complex.
3. **The group is too small for further splitting (controlled by min_samples_leaf)**:
   Restricting splitting when the number of samples in a node falls below a certain threshold helps prevent overfitting and results in smaller, more interpretable trees.

SUMMARY - Decision Tree Learning Algorithm

1. **Find the Best Split**:
   For each feature, evaluate all splits based on all possible thresholds and select the one that has the lowest impurity overall across all the features.
2. **Stop if Max Depth is Reached**:
   Stop the splitting process if the maximum allowable depth of the tree is reached.
3. **If LEFT is Sufficiently Large and Not Pure Repeat for LEFT**:
   If the left subset of data is both sufficiently large and not pure (contains more than one class), repeat the splitting process for the left subset.
4. **If RIGHT is Sufficiently Large and Not Pure Repeat for RIGHT**:
   Similarly, if the right subset of data is both sufficiently large and not pure, repeat the splitting process for the right subset.

DECISION TREE PARAMETER TUNING

If you look at the signature of the DecisionTreeClassifier class, it takes in many parameters like max_depth, min_samples_leaf, min_samples_split, max_features etc

```
class sklearn.tree.DecisionTreeClassifier(*, criterion='gini', splitter='best',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features=None, random_state=None,
max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None, ccp_alpha=0.0,
monotonic_cst=None)                                                    [sou
```

'Parameter tuning' means choosing parameters and assigning them values in a way that maximizes or minimizes a chosen performance metric (such as AUC or RMSE) on the validation set. In the case of AUC, our goal is to maximize it on the validation set by finding the parameter values that yield the highest score.

Some of the key parameters include:

- **criterion:** This parameter determines the impurity measure used for splitting. You can choose between 'gini' for Gini impurity and 'entropy' for information gain. The choice of criterion can significantly impact the quality of the splits in the decision tree.
- **max_depth:** This parameter controls the maximum depth of the decision tree. It plays a crucial role in preventing overfitting by limiting the complexity of the tree. Selecting an appropriate value for max_depth helps strike a balance between model simplicity and complexity.
- **min_samples_leaf:** This parameter specifies the minimum number of samples required in a leaf node (i.e. nodes at the bottom of the tree that decide the outcome). It influences the granularity of the splits. Smaller values can result in finer splits and a more complex tree, while larger values lead to coarser splits and a simpler tree.

By carefully tuning these parameters, you can find the right configuration that optimizes your model's performance, ensuring it's well-suited for your specific machine learning task.

You can select which parameters you want to use for tuning and for different values, see if the AUC improves or degrades. In our case, we will choose max_depth and max_samples_leaf

Selecting max_depth

To start the parameter tuning process, our initial focus will be on the 'max_depth' parameter. Our goal is to identify the optimal 'max_depth' value before proceeding to fine-tune other parameters. 'max_depth' governs the maximum depth of the decision tree. When set to 'None,' it imposes no restrictions, allowing the tree to grow as deeply as possible, potentially resulting in numerous layers.

We will try out different values of max_depth and check the AUC score

```
depths = [1, 2, 3, 4, 5, 6, 10, 15, 20, None]

for depth in depths:
    dt = DecisionTreeClassifier(max_depth=depth)
    dt.fit(X_train, y_train)

    # remember we need the column with negative scores
    y_pred = dt.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)

    print('%4s -> %.3f' % (depth, auc))
```

```
Output:
1 -> 0.606
2 -> 0.669
3 -> 0.739
4 -> 0.761
5 -> 0.767
6 -> 0.760
10 -> 0.706
15 -> 0.663
20 -> 0.654
None -> 0.657
```

We see that the optimal values appear to be around 76% for 'max_depth' values of 4, 5, and 6. This indicates that our best-performing tree should have between 4 to 6 layers. If there were no other parameters to consider, we could choose a depth of 4 to make the tree simpler, with only 4 layers instead of 5. A simpler tree is generally easier to read and understand, making it more transparent in terms of what's happening.

Selecting min_samples_leaf

Next, we move to tune "min_samples_leaf". Since we have determined max_depth to be 4-6, for each of these max_depth values, we will check for different "min_samples_leaf"

```
for depth in [4, 5, 6]:
    for s in [1, 5, 10, 15, 20, 500, 100, 200]:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_leaf=s)
        dt.fit(X_train, y_train)

        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        print('%4s, %4s -> %.3f' % (depth, s, auc))
```

```
4,    1 -> 0.761
4,    5 -> 0.761
4,   10 -> 0.761
4,   15 -> 0.764
4,   20 -> 0.761
4,  500 -> 0.680
4,  100 -> 0.756
4,  200 -> 0.747
5,    1 -> 0.767
5,    5 -> 0.768
5,   10 -> 0.762
5,   15 -> 0.772
5,   20 -> 0.774
5,  500 -> 0.680
5,  100 -> 0.763
5,  200 -> 0.759
6,    1 -> 0.749
6,    5 -> 0.762
6,   10 -> 0.778
6,   15 -> 0.785
6,   20 -> 0.774
6,  500 -> 0.680
6,  100 -> 0.776
6,  200 -> 0.768
```

To make the scores easier to comprehend, we create a dataframe with columns Max_depth, Min_Sample_Leaf, AUC

```
scores = []

for d in [4, 5, 6]:
    for s in [1, 2, 5, 10, 15, 20, 100, 200, 500]:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_leaf=s)
        dt.fit(X_train, y_train)

        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((d, s, auc))

columns = ['max_depth', 'min_samples_leaf', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)
df_scores.head()
```

```
df_scores.sort_values(by='auc', ascending=False).head()
```

	max_depth	min_samples_leaf	auc		max_depth	min_samples_leaf	Auc
0	4	1	0.655327	**22**	6	15	0.786997
1	4	2	0.697389	**4**	4	15	0.786521
2	4	5	0.712749	**13**	5	15	0.785398
3	4	10	0.762578	**14**	5	20	0.782159
4	4	15	0.786521	**23**	6	20	0.782120

From the sorted order, we can see that highest AUC of 78.7% is given by max_depth=6 & min_samples_leaf=15

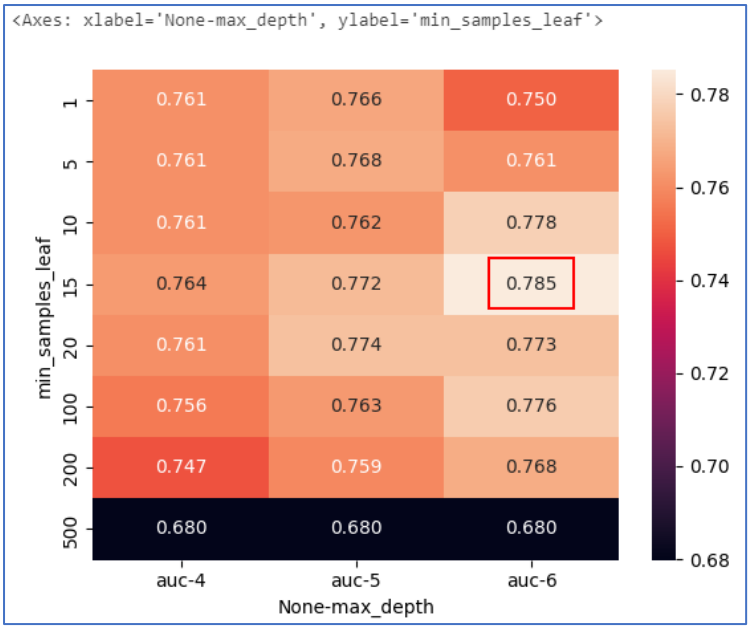
For a small dataset, this way of deducing the parameter tuned values works fine but for larger datasets, we want to visualize it through a heatmap where rows are min_sample_leaf, columns are max_depth and cells are the AUC values. For that, we first need to pivot the df_scores

```
df_scores_pivot = df_scores.pivot(index='min_samples_leaf', columns=['max_depth'], values=['auc'])
df_scores_pivot.round(3)
```

	auc		
max_depth	**4**	**5**	**6**
min_samples_leaf			
1	0.761	0.766	0.750
5	0.761	0.768	0.761
10	0.761	0.762	0.778
15	0.764	0.772	0.785
20	0.761	0.774	0.773
100	0.756	0.763	0.776
200	0.747	0.759	0.768
500	0.680	0.680	0.680

From the df_scores_pivot, we create heatmap using Seaborn

```
sns.heatmap(df_scores_pivot, annot=True, fmt=".3f")
```



In this heatmap, it's easy to identify the highest value as it appears the lightest, while the darkest shade represents the lowest or poorest value.

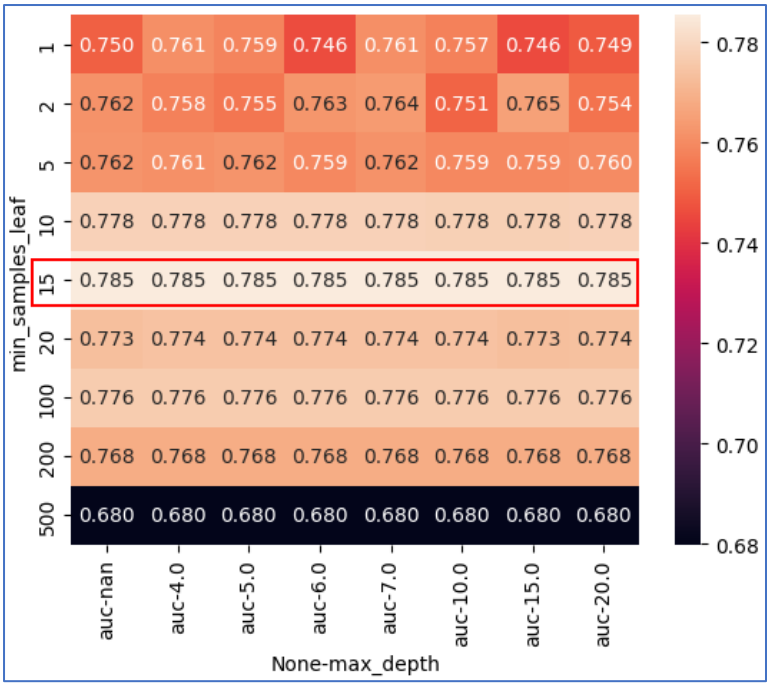
Constraints & Limitations

The above method of selecting the best parameter might not always be optimal. There's a possibility that a 'max_depth' of 7, 10, or another value works better, but we haven't explored those possibilities. This is because we initially tuned the 'max_depth' parameter and then selected the best 'min_samples_leaf.' for those depths only. For small datasets, it's feasible to try a variety of values, but with larger datasets, we need to constrain our search space to be more efficient. Therefore, it's often a good practice to first optimize the 'max_depth' parameter and then fine-tune the other parameters.

So, for this small dataset, lets try more values of max_depth

```
scores = []

for d in [4, 5, 6, 7, 10, 15, 20, None]:
    for s in [1, 2, 5, 10, 15, 20, 100, 200, 500]:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_leaf=s)
        dt.fit(X_train, y_train)

        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((d, s, auc))

columns = ['max_depth', 'min_samples_leaf', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)

df_scores.sort_values(by='auc', ascending=False).head()
```

	max_depth	min_samples_leaf	auc
40	10.0	15	0.785695
13	5.0	15	0.785471
67	NaN	15	0.785471
58	20.0	15	0.785319
31	7.0	15	0.785319



What we observe is max AUC is with min_samples_leaf = 15 and for all range of max_depth. From our experience, we do not want the tree to be very big by not specifying max_depth as that could lead to overfitting. So, we choose max_depth = 6 and with min_samples_leaf = 15

So, we train our **Final Decision Tree Model with max_depth = 6 and with min_samples_leaf = 15**

```
dt = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
dt.fit(X_train, y_train)
print(export_text(dt, feature_names=list(dv.get_feature_names_out())))
```

```
|--- records=no <= 0.50
|   |--- seniority <= 6.50
|   |   |--- amount <= 862.50
|   |   |   |--- price <= 925.00
|   |   |   |   |--- amount <= 525.00
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- amount >  525.00
|   |   |   |   |   |--- class: 1
|   |   |   |--- price >  925.00
|   |   |   |   |--- price <= 1382.00
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- price >  1382.00
|   |   |   |   |   |--- class: 0
|   |   |--- amount >  862.50
|   |   |   |--- assets <= 8250.00
|   |   |   |   |--- job=fixed <= 0.50
|   |   |   |   |   |--- assets <= 3425.00
|   |   |   |   |   |   |--- class: 1
|   |   |   |   |   |--- assets >  3425.00
|   |   |   |   |   |   |--- class: 1
|   |   |   |   |--- job=fixed >  0.50
|   |   |   |   |   |--- age <= 31.50
|   |   |   |   |   |   |--- class: 1
|   |   |   |   |   |--- age >  31.50
|   |   |   |   |   |   |--- class: 1
...
|   |   |   |   |   |--- class: 1
|   |   |--- assets >  7250.00
|   |   |   |--- class: 0
```

ENSEMBLE LEARNING - RANDOM FOREST

Random Forest is a technique for combining multiple decision trees. Before diving into Random Forest, we'll explore the concept of ensemble modeling, where multiple models act as a '**board of experts**.'

When a client approaches a bank for a loan, they submit an application with basic information, and features are extracted from this application. These features are then fed into a decision tree, which provides a score representing the probability of the customer defaulting on the loan. Based on this score, the bank makes a lending decision.

Now, let's imagine an alternative scenario where we don't use a decision tree but rather rely on a 'board of experts,' consisting of five experts. When a customer submits an application, it's distributed to each of these experts. Each expert independently evaluates the application and decides whether to approve or reject it. The final decision is determined by a majority vote—if the majority of experts say 'yes,' the bank approves the loan; if the majority says 'no,' the application is rejected.
The underlying concept of the 'board of experts' is based on the belief that the collective wisdom of five experts is more reliable than relying solely on one expert's judgment. By aggregating multiple expert opinions, we aim to make better decisions.

This same concept can be applied to models. Instead of a 'board of five experts,' we can have five models (g1, g2, …, g5), each of which returns a probability of default. We can then aggregate these model predictions by calculating the average: $(1/n) * \Sigma(p_i)$.

Probability of Board of Models = Avg of probabilities of all models

This method of aggregating multiple models is applicable to any type of model. Specifically, when we **ensemble decision trees, we create what is known as a 'Random Forest.'**

Why called "RANDOM FOREST"?

Why do we call it a 'random forest' and not just a 'forest'? The reason is that if we take the same application (with the same set of features) and build the same group of trees with identical parameters, the resulting trees would also be identical. These identical trees would produce exactly the same probability of default, and therefore, the average would be the same as well. Essentially, we would be training the same model five times, which is pointless.

In a random forest, each of the applications or features that the trees receive is slightly different i.e. each model gets a random subset of features. For instance, if we have a total of 10 features, each tree might receive any random 7 out of the 10 features, creating a distinct set for each tree. Logically, it needs to select any random 7 out of 10 features which can be done in ${}^{10}C_7$ ways = 120 ways.

To illustrate this with a smaller example, let's consider 3 features: assets, debt, and price. We need to select 2 features for each tree which can be done in ${}^{3}C_2$ ways = 3 ways. If we train only 3 models, we might have the following feature sets for each:

- Decision Tree #1: Features – assets, debt
- Decision Tree #2: Features – assets, price
- Decision Tree #3: Features – debt, price

Each would predict a probability. To obtain the final prediction, we calculate the average score as (1/3) * (p1 + p2 + p3).

Using scikitlearn for Random forest

```
from sklearn.ensemble import RandomForestClassifier

# n_estimators - number of models we want to use
rf = RandomForestClassifier(n_estimators=10)
rf.fit(X_train, y_train)

y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
# Output: 0.781835024581628

rf.predict_proba(X_val[[0]])
# Output: array([[1., 0.]])
```

- We import RandomForestClassifier class from sklearn.ensemble package.
- Create object of RandomForestClassifier with number of models (estimators). Default is 100
- Train the model on training data using rf.fit(X_train, y_train)
- Predict the probability of Positive Class of Validation data using rf.predict_proba(X_val)
- Compute AUC Score

This model achieves an AUC score of 78.2%, which is relatively good and on par with the best decision tree model without any specific tuning. In this instance, we used the default hyperparameters and only reduced the 'n_estimators' value from the default of 100 to 10.

random_state parameter - it's important to recognize that a random forest introduces an element of randomness during training. Consequently, when we retrain the model and make predictions again, we may obtain slightly different results due to this randomization. To ensure consistent and reproducible results, we can use the 'random_state' parameter. By setting a fixed 'random_state,' regardless of how many times we run the model, the results will remain the same.

```
rf = RandomForestClassifier(n_estimators=10, random_state=1)
rf.fit(X_train, y_train)

y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
# Output: 0.7744726453706618

rf.predict_proba(X_val[[0]])
# Output: array([[0.9, 0.1]])
```

Tuning Random Forest

```
class sklearn.ensemble.RandomForestClassifier(n_estimators=100, *,
criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='sqrt', max_leaf_nodes=None,
min_impurity_decrease=0.0, bootstrap=True, oob_score=False, n_jobs=None,
random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0,
max_samples=None, monotonic_cst=None)                                    [so
```

Since Random Forest consists of multiple Decision Trees, the parameters for tuning are same as for Decision Trees like max_depth, min_samples_leaf etc. Additional parameter is the number of estimators (no of Decision Trees)

Tuning for n_estimators

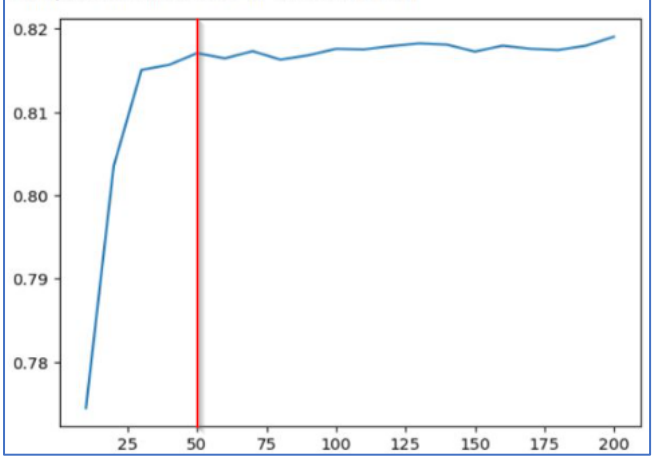
Lets start with tuning for n_estimators. We iterate over different values of n and tabulate the AUC scores for each n. n = 10, 20, 30, ….190, 200. **This takes about 20 secs**

```
scores = []

for n in range(10, 201, 10):
    rf = RandomForestClassifier(n_estimators=n, random_state=1)
    rf.fit(X_train, y_train)

    y_pred = rf.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)

    scores.append((n, auc))

df_scores = pd.DataFrame(scores, columns=['n_estimators', 'auc'])
df_scores
```

	n_estimators	auc
0	10	0.774473
1	20	0.803532
2	30	0.815075
3	40	0.815686
4	50	0.817082
5	60	0.816458
6	70	0.817321
7	80	0.816307
8	90	0.816824
9	100	0.817599
10	110	0.817527
11	120	0.817939
12	130	0.818253
13	140	0.818102
14	150	0.817270
15	160	0.817981
16	170	0.817606
17	180	0.817463
18	190	0.817981
19	200	0.819050

```
# x-axis - n_estimators
# y-axis - auc score
plt.plot(df_scores.n_estimators, df_scores.auc)
```

```
[<matplotlib.lines.Line2D at 0x7f2343e9f110>]
```



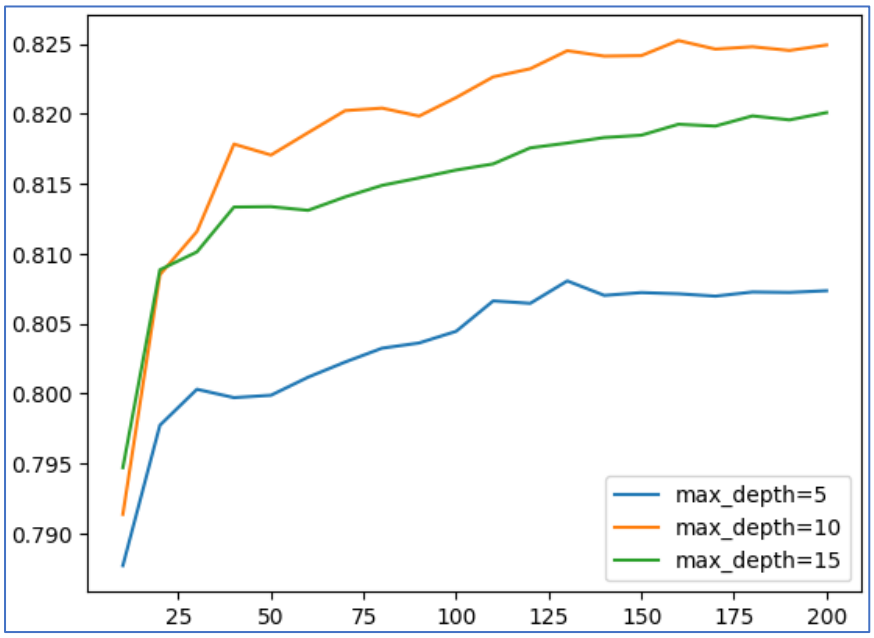
Observe that the model's performance improves as we increase the number of estimators from 10 up to 50, but beyond 50, it reaches a plateau meaning additional trees don't significantly enhance the performance. Hence, training more than 50 trees doesn't appear to be beneficial.

Tuning for max_depth

For different max_depth d of 5, 10, 15, we will also iterate over different n from 10 to 200. **This takes about 45 seconds**

```
scores = []

for d in [5, 10, 15]:
    for n in range(10, 201, 10):
        rf = RandomForestClassifier(n_estimators=n,
                                    max_depth=d,
                                    random_state=1)
        rf.fit(X_train, y_train)

        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((d, n, auc))

columns = ['max_depth', 'n_estimators', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)
```

```
# Let's plot it
for d in [5, 10, 15]:
    df_subset = df_scores[df_scores.max_depth == d]

    plt.plot(df_subset.n_estimators, df_subset.auc,
             label='max_depth=%d' % d)

plt.legend()
```




```
# Let's select 10 as the best value
max_depth = 10
```

Observe that the AUC scores for 'max_depth' values of 10 and 15 are initially quite close, but after a certain point, the score for 'max_depth' 15 begins to level off, showing only marginal improvement. In contrast, 'max_depth' 10 continues to perform significantly better, peaking at around n_estimators=125. This clearly illustrates that the choice of 'max_depth' indeed matters. We can confidently select a value of 10 as the best choice, as the difference between 10 and 15, and between 10 and 5, is notably significant.

So, we fix max_depth = 10.

Tuning for min_samples_leaf

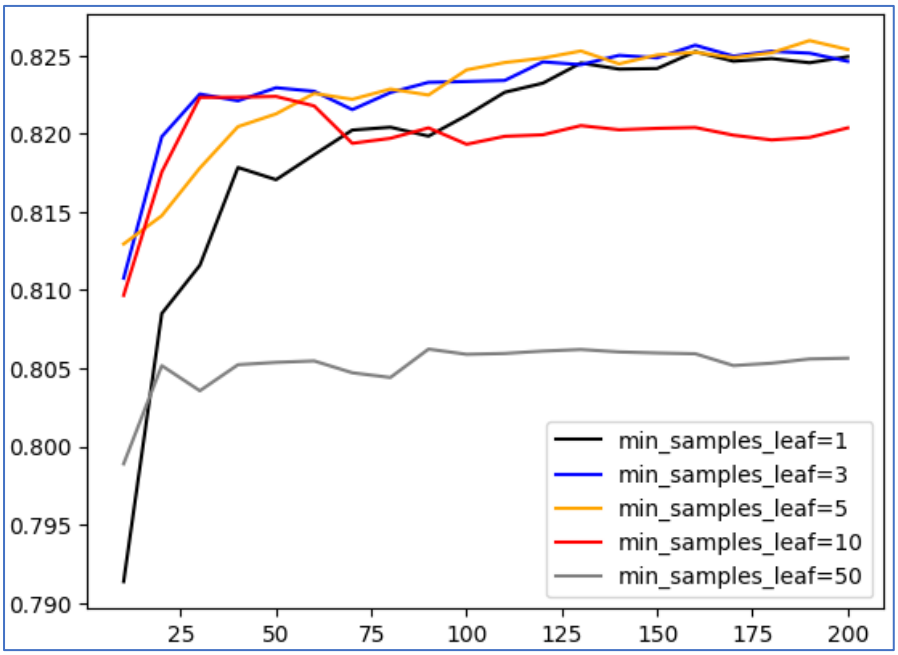
After fixing max_depth = 10, we replace the iterator of max_depth with that of Sample_leaf s = 1,3,5,10,50. This takes about 60 secs to execute

```
scores = []

for s in [1, 3, 5, 10, 50]:
    for n in range(10, 201, 10):
        rf = RandomForestClassifier(n_estimators=n,
                                    max_depth=max_depth,
                                    min_samples_leaf=s,
                                    random_state=1)
        rf.fit(X_train, y_train)

        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)

        scores.append((s, n, auc))

columns = ['min_samples_leaf', 'n_estimators', 'auc']
df_scores = pd.DataFrame(scores, columns=columns)
```

Since the default colors on the Plot are not distinguishable, we specify custom colors corresponding to leaf value and iterate over the tuple (leaf_value, color). Tuple is created using the zip function

```
colors = ['black', 'blue', 'orange', 'red', 'grey']
min_samples_leaf_values = [1, 3, 5, 10, 50]
list(zip(min_samples_leaf_values, colors))

# Output: [(1, 'black'), (3, 'blue'), (5, 'orange'), (10, 'red'), (50, 'grey
```

```
colors = ['black', 'blue', 'orange', 'red', 'grey']
min_samples_leaf_values = [1, 3, 5, 10, 50]

for s, col in zip(min_samples_leaf_values, colors):
    df_subset = df_scores[df_scores.min_samples_leaf == s]

    plt.plot(df_subset.n_estimators, df_subset.auc,
             color=col,
             label='min_samples_leaf=%d' % s)

plt.legend()
```



Observe that the three most favorable options are 1, 3, and 5. Among these, ‘min_samples_leaf’ of 3 stands out a good choice since it achieves good performance earlier than the others.

Other tuning parameters

other useful parameters to consider:

- **max_features:** This parameter determines how many features each decision tree receives during training. It's essential to remember that random forests work by selecting only a subset of features for each tree.
- **bootstrap:** Bootstrap introduces another form of randomization, but at the row level. This randomization ensures that the decision trees are as diverse as possible.
- **n_jobs:** number of jobs to run in parallel for fit(), predict(), etc. The training of decision trees can be parallelized because all the models are independent of each other. Default is None meaning no parallelization. Set it to -1 to use all processors parallely

Final tuned Random Forest model

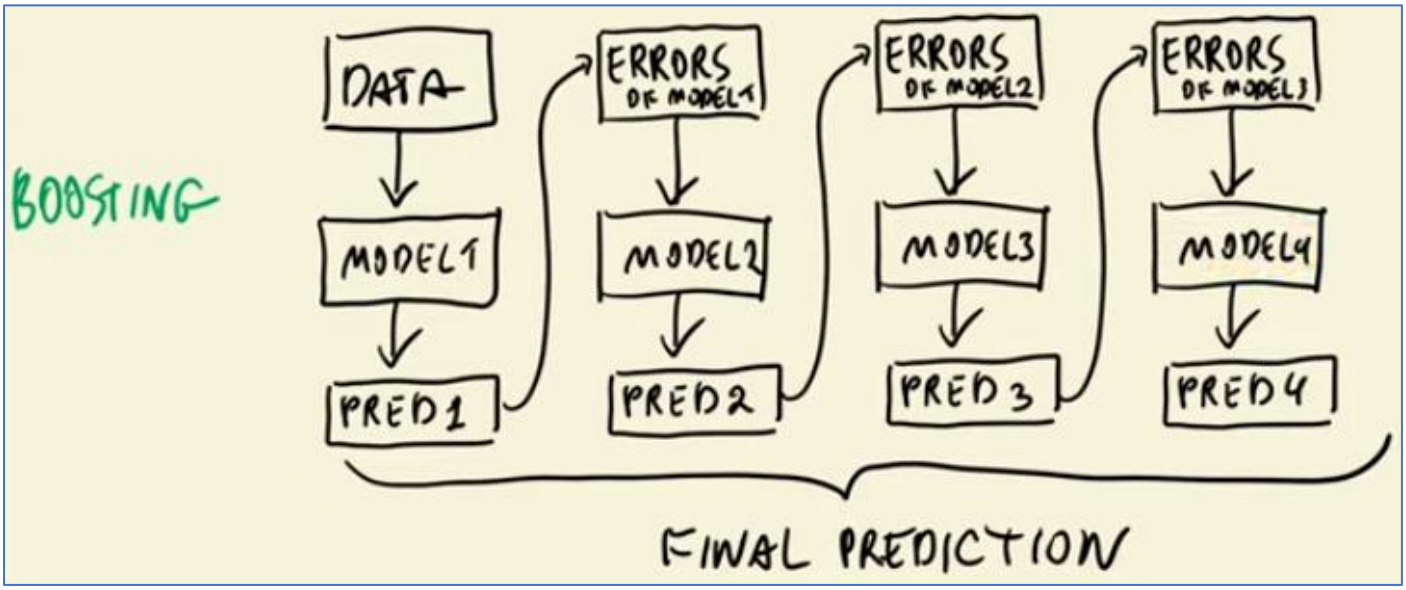
max_depth = 10, min_samples_leaf = 3, n_estimators = 100, n_jobs = -1

```
rf = RandomForestClassifier(n_estimators=100,
                            max_depth=max_depth,
                            min_samples_leaf=min_samples_leaf,
                            random_state=1,
                            n_jobs=-1)
rf.fit(X_train, y_train)

# Output:
# RandomForestClassifier(max_depth=10, min_samples_leaf=3, n_jobs=-1,
#                        random_state=1)
```

GRADIENT BOOSTING - XGBOOST

Concept of Boosting

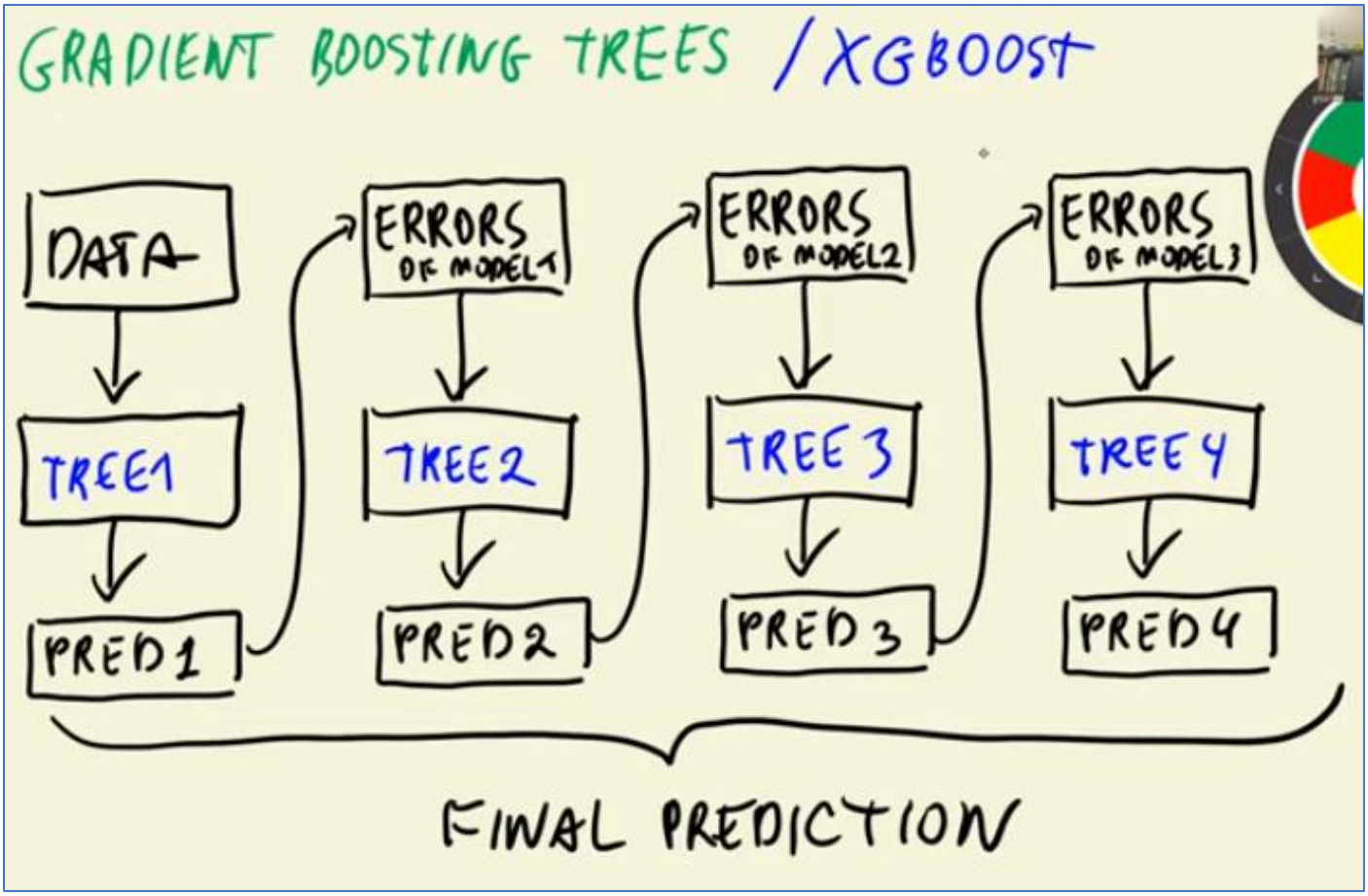
Boosting is a different way of combining models where we train Model 1 on the dataset and it makes Prediction 1. We extract the errors Model 1 made i.e. Error 1 and feed that to train Model 2 to correct those errors and make Predictions 2. We extract errors of Model 2 i.e. Errors 2 and feed that to train Model 3 to correct those errors and make Predictions 3. This process can be repeated for multiple iterations. At the end of these iterations, we combine the predictions from these multiple models into the final prediction.



The core idea behind boosting is the **sequential training of multiple models**, where each subsequent model corrects the mistakes of the previous one.

GRADIENT BOOSTING TREE - XGBOOST

If you replace all the models in the above diagram with Decision Trees, you get **GRADIENT BOOSTING TREES**. And XGBoost is the library that you use to implement Gradient Boosting Tree



Difference between Random Forest & Gradient Boosting Tree-

In a random forest, multiple independent decision trees are trained on the same dataset. The decision trees are independent of each other. The final prediction is achieved by aggregating the results of these individual trees, typically by taking an average: $((1/n) * \Sigma(p_i))$.

While in Boosting, the decision trees are sequentially trained on Errors from the previous trees. There is dependency between the trees. Final prediction is the weighted average of the results of the individual trees

XGBoost library is installed in Python using pip install xgboost

```
!pip install xgboost

import xgboost as xgb
```

Training the first model

The first step in the process is to structure the training data into a specialized data format known as '**DMatrix**.' This format is optimized for training XGBoost models, allowing for faster training.

```
features = list(dv.get_feature_names_out())
dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=features)
dval = xgb.DMatrix(X_val, label=y_val, feature_names=features)
```

We create two D-matrix, dtrain & dval

Then we create a dictionary xgb_params to define the parameters for the XGBoost

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'nthread': 8,

    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200)
```

See https://xgboost.readthedocs.io/en/latest/parameter.html for more info on parameters. Some of the most crucial parameters include:

- **eta:** This parameter represents the learning rate, determining how quickly the model learns. Its value is between 0 and 1
- **max_depth:** Similar to random forests and decision trees, 'max_depth' controls the size of the trees. how many levels deep
- **min_child_weight:** This parameter controls the minimum number of observations that should be present in a leaf node, similar to the 'min_samples_leaf' in decision trees.
- **objective:** Xgboost can be used for both Classification & Regression and it can be optimized by specifying correct purpose. In this case, we want to do Binary Classification using Logistic Regression
- **nthread:** XGBoost has the capability to parallelize training, and here, we specify how many threads to utilize.
- **seed:** This parameter controls the randomization used in the model.
- **verbosity:** It allows us to control the level of detail in the warnings and messages generated during training. 0=No messages, 1=Warnings, Errors, 2=Verbose(will output all the processing of the XGBoost)

And, then use the xgb.train() function passing in the xgb_params, dtrain Training DMatrix & setting number of boosting iterations (num_boost_round=200)

Then, to test the model against Validation, we use predict() function to return predictions

```
y_pred = model.predict(dval)
y_pred
```

✓ 0.2s

```
array([3.43261547e-02, 6.90845598e-04, 3.03691253e-03, 9.81778800e-02,
       8.90289521e-05, 1.03313965e-03, 1.35689916e-03, 2.69821793e-01,
       3.84416819e-01, 8.63077512e-05, 7.33645633e-02, 9.92994249e-01,
```

And AUC Score is 80%

```
roc_auc_score(y_val, y_pred)
```

✓ 0.0s

```
0.8072491584122448
```

Which is a good score as we used only default parameters

If we change num_boost_round to 10, then AUC increases is 81.5%.

```
model = xgb.train(xgb_params, dtrain, num_boost_round=10)
```

```
roc_auc_score(y_val, y_pred)
```

```
0.8152745150274878
```

Higher num_boost_round can lead to Overfitting and hence AUC drops

Monitor XGBoost Training Performance

In **XGBoost**, it's feasible to monitor the performance of the training process, allowing us to closely observe each stage of the training procedure. To achieve this, after each iteration where a new tree is trained, we can promptly evaluate its performance on our validation data to assess the results. For this purpose, we can establish a watchlist that comprises the datasets we intend to use for evaluation.

```
watchlist = [(dtrain, 'train'), (dval, 'val')]
```

We add this watchlist as "evals" parameter to xgb.train() function. By default, XGBoost displays the error rate (logloss) but we want to see the AUC for each iteration

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  evals=watchlist)

# Output:
# [0]   train-logloss:0.49703   val-logloss:0.54305
# [1]   train-logloss:0.44463   val-logloss:0.51462
# [2]   train-logloss:0.40707   val-logloss:0.49896
# [3]   train-logloss:0.37760   val-logloss:0.48654
# [4]   train-logloss:0.35990   val-logloss:0.48007
# [5]   train-logloss:0.33931   val-logloss:0.47563
# [6]   train-logloss:0.32586   val-logloss:0.47413
```

So, we specify "eval_metric" parameter to xgb.train() function

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  evals=watchlist)

# Output:
# [0]   train-auc:0.86730   val-auc:0.77938
# [1]   train-auc:0.89140   val-auc:0.78964
# [2]   train-auc:0.90699   val-auc:0.79010
# [3]   train-auc:0.91677   val-auc:0.79967
```

```
[104]   train-auc:0.99999   val-auc:0.80637
[105]   train-auc:0.99999   val-auc:0.80603
[106]   train-auc:0.99999   val-auc:0.80551
[107]   train-auc:1.00000   val-auc:0.80517
[108]   train-auc:1.00000   val-auc:0.80491
[109]   train-auc:1.00000   val-auc:0.80550
```

This gives the AUC when predicted using Training data & Validation data. AUC for both increases with each iteration. But, after a certain number of iterations, AUC for Training data reaches 1 and stays constant at 1 while the AUC for Validation data peaks and then starts decreasing. This point denotes that the Model has started Overfitting the data

To make this output more user-friendly, it would be beneficial to visualize it. Instead of printing output for every iteration, we can use verbose_eval=5 to display results only for every 5th iteration, making the monitoring process more manageable.

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)

# Output:
# [0]       train-auc:0.86730   val-auc:0.77938
# [5]       train-auc:0.93086   val-auc:0.80858
# [10]      train-auc:0.95447   val-auc:0.80851
# [15]      train-auc:0.96554   val-auc:0.81334
# [20]      train-auc:0.97464   val-auc:0.81729
# [25]      train-auc:0.97953   val-auc:0.81686
# [30]      train-auc:0.98579   val-auc:0.81543
```

Parsing the XGBoost Monitoring Output

XGBoost doesn't provide an easy way to extract the above information since it's printed to standard output. However, in Jupyter Notebook, there's a method to capture whatever is printed to standard output and manipulate it. You can use the command %%capture output to achieve this. It captures all the content that the code outputs into a special object, which you can then use to extract the information.

```
%%capture output

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)


s = output.stdout

print(s)
# Output:
# [0]       train-auc:0.86730   val-auc:0.77938
# [5]       train-auc:0.93086   val-auc:0.80858
# [10]      train-auc:0.95447   val-auc:0.80851
# [15]      train-auc:0.96554   val-auc:0.81334
```

Now that we have the captured output in a string, the first step is to split it into individual lines by using the new line operator \n. The result is a string for each line of the output.

```
s.split('\n')

# Output:
# ['[0]\ttrain-auc:0.86730\tval-auc:0.77938',
#  '[5]\ttrain-auc:0.93086\tval-auc:0.80858',
#  '[10]\ttrain-auc:0.95447\tval-auc:0.80851',
#  '[15]\ttrain-auc:0.96554\tval-auc:0.81334',
```

Each line consists of three components: the number of iterations, the evaluation on the training dataset, and the evaluation on the validation dataset. We can split these components using the tabulator operator \t, resulting in three separate components. To ensure the correct format (integer, float, float), we utilize the strip method and perform the necessary string-to-integer and string-to-float conversions.

```
line = s.split('\n')[0]
line
# Output: '[0]\ttrain-auc:0.86730\tval-auc:0.77938'

line.split('\t')
# Output: ['[0]', 'train-auc:0.86730', 'val-auc:0.77938']

num_iter, train_auc, val_auc = line.split('\t')
num_iter, train_auc, val_auc
# Output: ('[0]', 'train-auc:0.86730', 'val-auc:0.77938')

int(num_iter.strip('[]'))
# Output: 0
float(train_auc.split(':')[1])
# Output: 0.8673
float(val_auc.split(':')[1])
# Output: 0.77938
```

We can combine all these steps to transform the information (number of iterations, AUC on the training data, and AUC on the validation data) from the output into a dataframe. The following snippet encapsulates all these steps within a single function for ease of use. This allows us to plot the data and perform further analysis.

```
def parse_xgb_output(output):
    results = []

    for line in output.stdout.strip().split('\n'):
        it_line, train_line, val_line = line.split('\t')

        it = int(it_line.strip('[]'))
        train = float(train_line.split(':')[1])
        val = float(val_line.split(':')[1])

        results.append((it, train, val))

    columns = ['num_iter', 'train_auc', 'val_auc']
    df_results = pd.DataFrame(results, columns=columns)
    return df_results
```

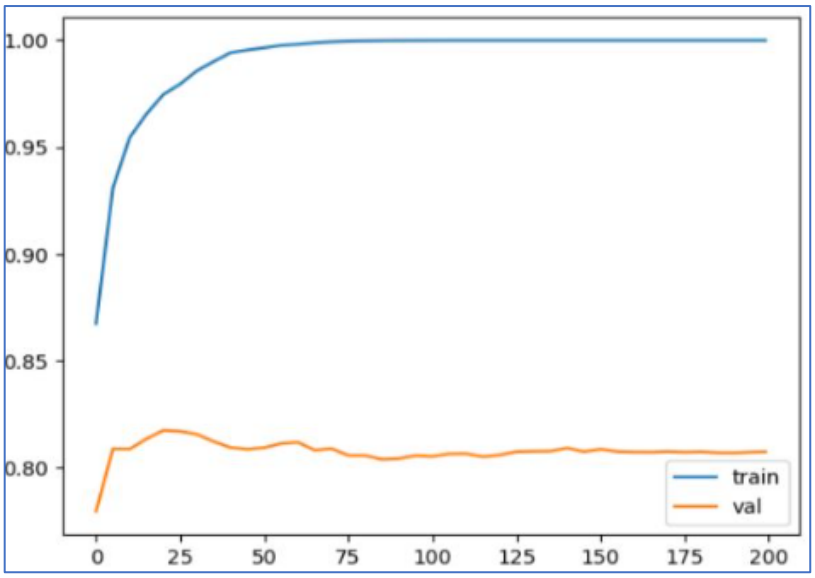
```
df_score = parse_xgb_output(output)
df_score
```

	num_iter	train_auc	val_auc
0	0	0.86730	0.77938
1	5	0.93086	0.80858
2	10	0.95447	0.80851
3	15	0.96554	0.81334
4	20	0.97464	0.81729
5	25	0.97953	0.81686
6	30	0.98579	0.81543
7	35	0.99011	0.81206

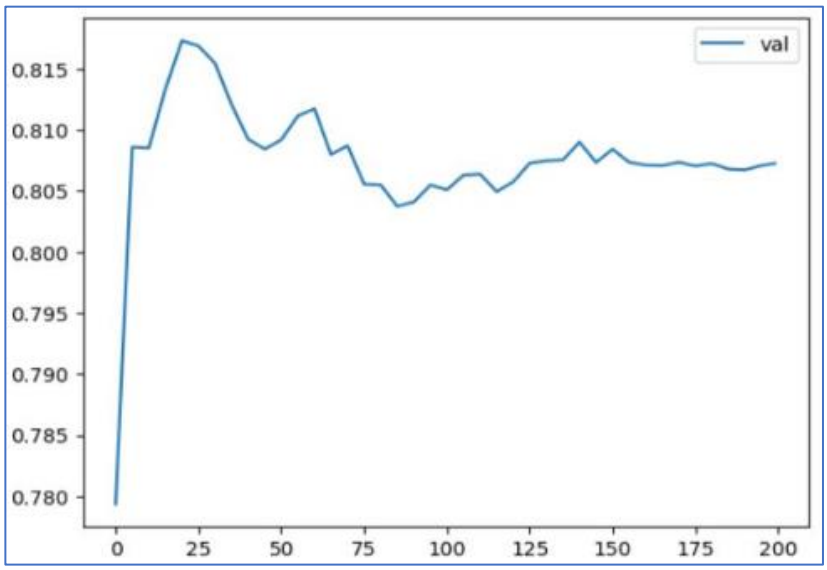
```
# x-axis - number of iterations
# y-axis - auc
plt.plot(df_score.num_iter, df_score.train_auc, label='train')
plt.plot(df_score.num_iter, df_score.val_auc, label='val')
plt.legend()
```

```
plt.plot(df_score.num_iter, df_score.val_auc, label='val')
plt.legend()
```

	num_iter	train_auc	val_auc
8	40	0.99421	0.80922
9	45	0.99548	0.80842
10	50	0.99653	0.80918
11	55	0.99765	0.81114
12	60	0.99817	0.81172
13	65	0.99887	0.80798
14	70	0.99934	0.80870
15	75	0.99965	0.80555
16	80	0.99979	0.80549
17	85	0.99988	0.80374
18	90	0.99993	0.80409
19	95	0.99996	0.80548
20	100	0.99998	0.80509
21	105	0.99999	0.80629
22	110	1.00000	0.80637
23	115	1.00000	0.80494
24	120	1.00000	0.80574
25	125	1.00000	0.80727
26	130	1.00000	0.80746
27	135	1.00000	0.80753
28	140	1.00000	0.80899
29	145	1.00000	0.80733
30	150	1.00000	0.80841
31	155	1.00000	0.80734
32	160	1.00000	0.80711
33	165	1.00000	0.80707
34	170	1.00000	0.80734
35	175	1.00000	0.80704
36	180	1.00000	0.80723
37	185	1.00000	0.80678
38	190	1.00000	0.80672
39	195	1.00000	0.80708
40	199	1.00000	0.80725



Observe that the AUC on the training dataset consistently improves. However, the picture is different for the validation dataset. The curve reaches its peak earlier and then starts to decline and stagnate, indicating the onset of **overfitting**. This decline in performance on the validation dataset is more apparent when plotting only the AUC on validation, while the AUC on the training dataset remains consistently high.



The decline in performance is more evident when we exclusively plot the validation graph.

Tuning the XGBoost Model

We will tune the following parameters-

- eta (Learning Rate)
- max_depth
- max_child_weight (similar to max_samples_leaf)

Tuning eta

Eta, also known as the learning rate, determines the influence of the following model when correcting the results of the previous model. If the weight is set to 1.0, all new predictions are used to correct the previous ones. However, when the weight is 0.3, only 30% of the new predictions are considered. In essence, eta governs the size of the steps taken during the learning process.

Let's explore how different values of eta impact model performance. For this, we'll create a dictionary called 'scores' to store the performance scores for each value of eta. The dict socres would have key "eta=<>" and value = dataframe of monitoring for that eta

For eta = 0.3

```
scores = {}
```

We train the model using eta = 0.3 and capture the output

```
%%capture output

xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

We set the key = "eta=0.3" and value as parse_xgb_output() which is dataframe that contains train_auc and val_auc values for different num_iter.

```
key = 'eta=%s' % (xgb_params['eta'])
scores[key] = parse_xgb_output(output)
key

# Output: 'eta=0.3'
```

```
scores

# Output:
# {'eta=0.3':     num_iter  train_auc  val_auc
#  0           0    0.86730  0.77938
#  1           5    0.93086  0.80858
#  2          10    0.95447  0.80851
#  3          15    0.96554  0.81334
#  4          20    0.97464  0.81729
#  5          25    0.97953  0.81686
#    ...
#  36        180    1.00000  0.80723
#  37        185    1.00000  0.80678
#  38        190    1.00000  0.80672
#  39        195    1.00000  0.80708
#  40        199    1.00000  0.80725}
```

For other eta = 1.0, 0.1, 0.05, 0.01

```
%%capture output

xgb_params = {
    'eta': 1.0,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

```
%%capture output

xgb_params = {
    'eta': 0.1,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

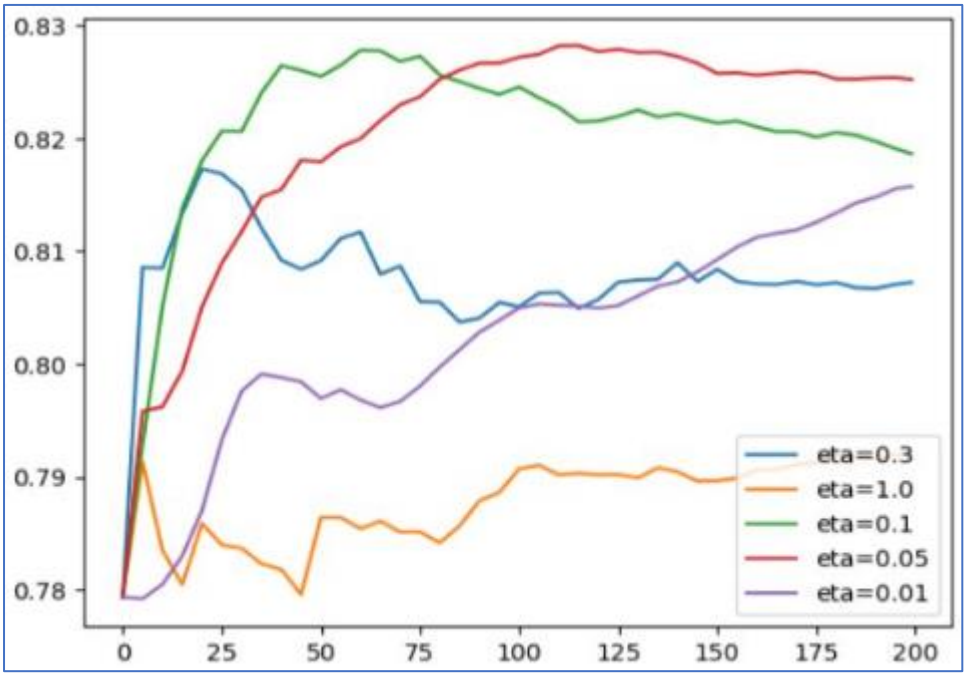
```
%%capture output

xgb_params = {
    'eta': 0.05,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

```
%%capture output

xgb_params = {
    'eta': 0.01,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

So, now we have 5 keys in scores dict and we plot the info for each eta

```
scores.keys()
# Output: dict_keys(['eta=0.3', 'eta=1.0', 'eta=0.1', 'eta=0.05', 'eta=0.01'
```

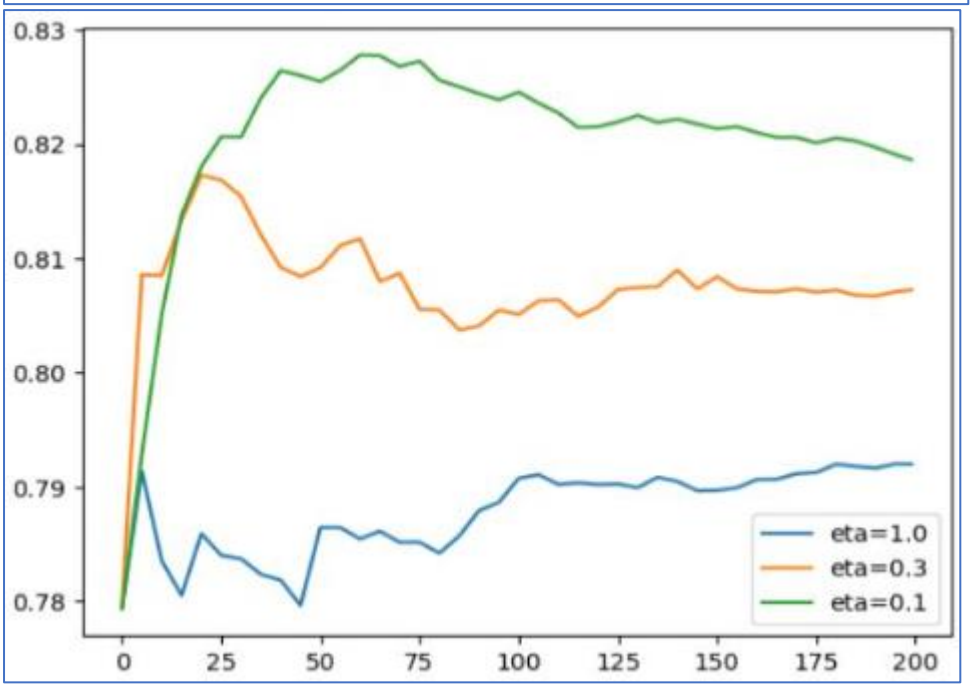
```
for key, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=key)
plt.legend()
```



Inference

First, concentrate on plots of ‘eta=1.0’, ‘eta=0.3’, and ‘eta=0.1’.

```
etas = ['eta=1.0', 'eta=0.3', 'eta=0.1']
for eta in etas:
    df_score = scores[eta]
    plt.plot(df_score.num_iter, df_score.val_auc, label=eta)
plt.legend()
```

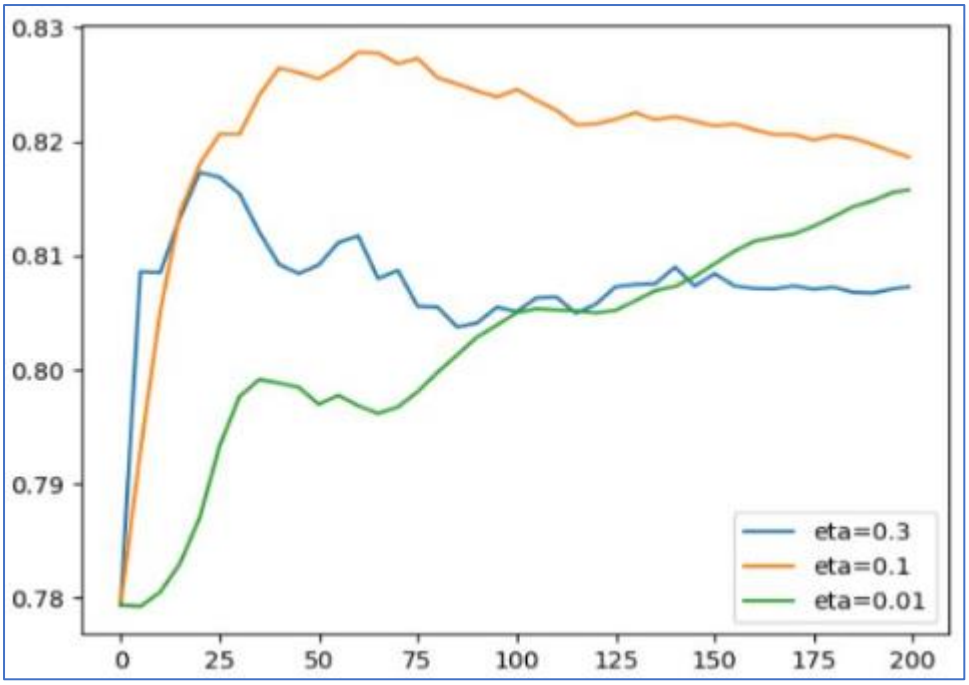


Learning Rate is depicted by the slope as num_iterations increase from 0 to the right. The higher the value of eta, sharper the slope. 'eta=1.0' exhibits the worst performance. It quickly reaches peak performance but then experiences a sharp decline, maintaining a consistently poor level. 'eta=0.3' performs reasonably well until around iteration 25, after which it steadily deteriorates. On the other hand, 'eta=0.1' demonstrates a slower growth rate, reaching its peak at a later stage before descending. This pattern is a direct reflection of the learning rate's influence.

The learning rate controls both the speed at which the model learns and the size of the steps it takes during each iteration. If the steps are too large, the model learns rapidly but eventually starts to degrade due to the excessive step size, resulting in **overfitting**. Conversely, a smaller learning rate signifies slower but more stable learning. Such models tend to degrade more gradually, and their overfitting tendencies are less pronounced compared to models with higher learning rates.

Next let's look at eta=0.3, eta=0.1, and eta=0.01

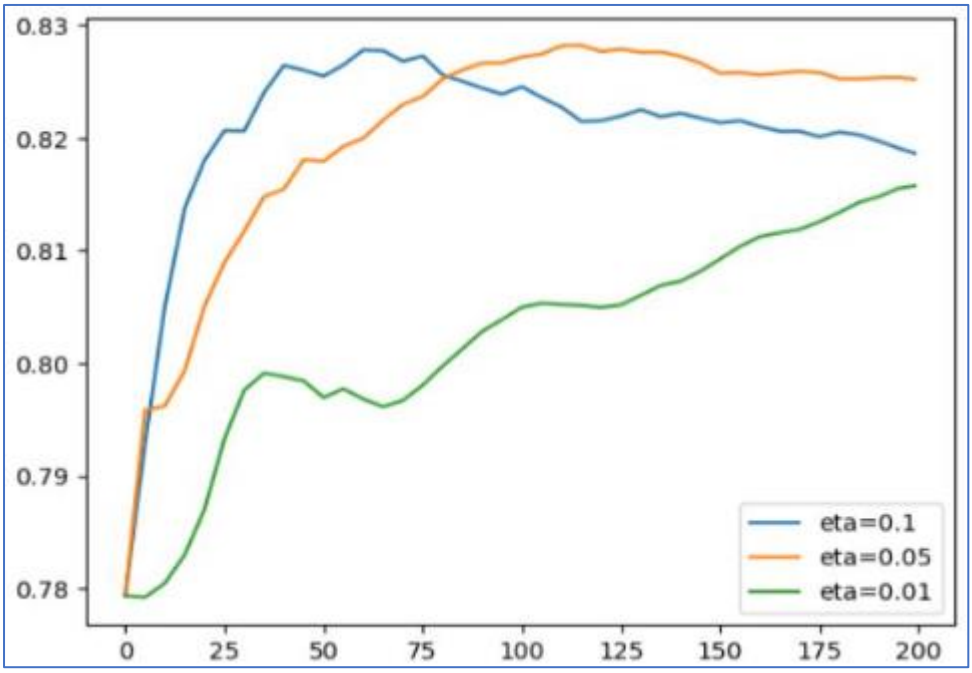
```
etas = ['eta=0.3', 'eta=0.1', 'eta=0.01']
for eta in etas:
    df_score = scores[eta]
    plt.plot(df_score.num_iter, df_score.val_auc, label=eta)
plt.legend()
```



'eta=0.01' displays an extremely slow learning rate, making it challenging to estimate how long it might take to outperform the other model (represented by the orange curve). This model's progress is painstakingly slow, as the steps it takes are exceedingly tiny.
On the other hand, 'eta=0.3' takes a few significant steps initially but succumbs to overfitting more rapidly. In this plot, 'eta=0.1' seems to strike the ideal balance, particularly between 50 and 75 iterations. It may take a bit longer to reach its peak performance, but the resulting performance improvement justifies the wait.

There was also eta=0.05 let's finally look also at this plot.

```
etas = ['eta=0.1', 'eta=0.05', 'eta=0.01']
for eta in etas:
    df_score = scores[eta]
    plt.plot(df_score.num_iter, df_score.val_auc, label=eta)
plt.legend()
```



The 'eta=0.05' model requires approximately twice as many iterations to converge when compared to the blue model ('eta=0.1'). Although it takes smaller steps and requires more time, the end result is still inferior to the blue model. Thus, it's evident that the 'eta=0.1' model stands out as the best option, as it achieves better performance with fewer steps. So, **eta = 0.1 for final model**

Tuning max_depth

Now, we set eta = 0.1 and tune for max_depth = [6,3,4,10]. We reset the scores dict to keep track of the new experiments with max_depth

```
scores = {}


%%capture output

xgb_params = {
    'eta': 0.1,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)


key = 'max_depth=%s' % (xgb_params['max_depth'])
scores[key] = parse_xgb_output(output)
key
```

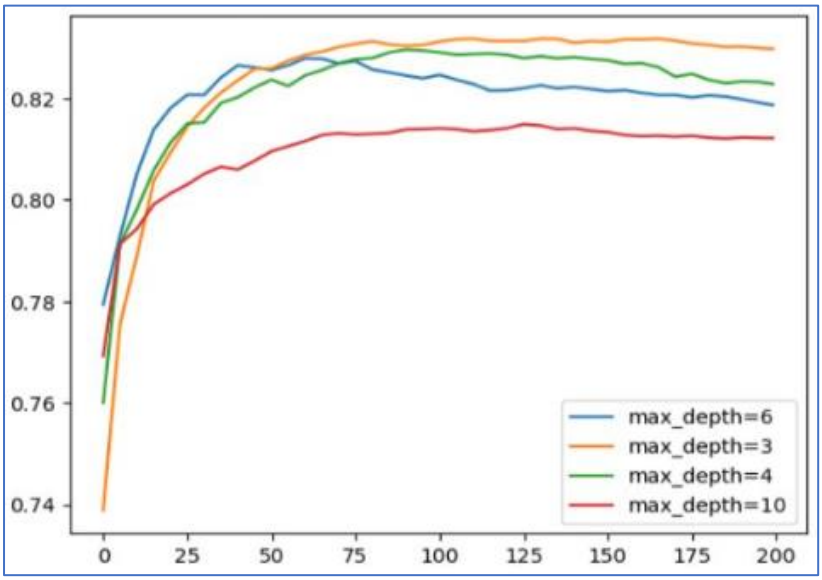
```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 4,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 10,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

We plot the Val_auc score vs num_iterations for each max_depth

```
for max_depth, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=max_depth)

#plt.ylim(0.8, 0.84)
plt.legend()
```



Depth of 10 is worst. depth = 6 is the second worst. **'max_depth' = 3 is the best depth.**

Tuning min_child_weight

We'll set 'eta' to 0.1 and 'max_depth' to 3. We reset the scores dict to keep track of the new experiments with min_child_weight = [1, 10, 30]

```
scores = {}


%%capture output

xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)


key = 'min_child_weight=%s' % (xgb_params['min_child_weight'])
scores[key] = parse_xgb_output(output)
key

# Output: 'min_child_weight=1'
```

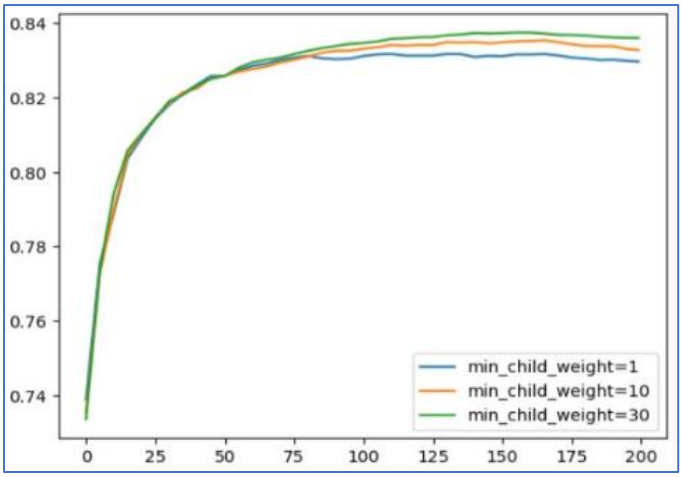
```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 10,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

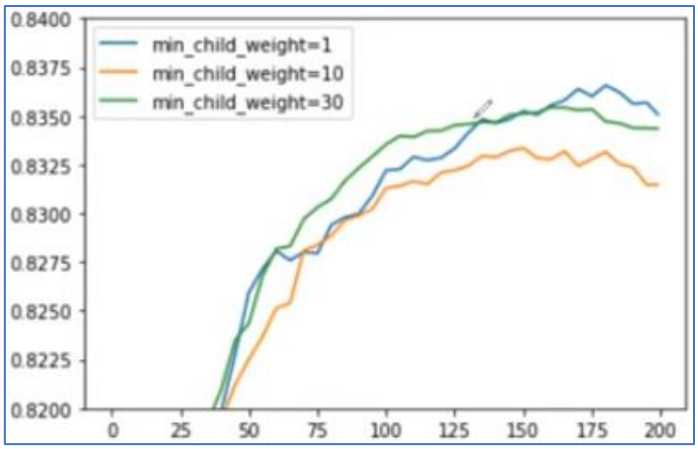
```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 30,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=200,
                  verbose_eval=5,
                  evals=watchlist)
```

Lets plot the graphs for different min_child_weight

```
for min_child_weight, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=min_child_weight)

plt.legend()
```

```
for min_child_weight, df_score in scores.items():
    plt.plot(df_score.num_iter, df_score.val_auc, label=min_child_weight)

plt.ylim(0.8, 0.84)
plt.legend()
```





Here, even after trying out different min_child_weight values, the plots are very close. Also, for min_child_weight = 1, it maintains the AUC even after num_iterations >175. So, we select **min_child_weight = 1**

Train final XGBoost Model

To train the final model, we need to determine the number of iterations for training. From the tuning of eta, max_depth, min_child_weight, we see good performance for number of iterations <175. So, we set num_boost_round = 175

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=175)
```

Tips for tuning

Always creating all these plots may not be necessary. You can also examine the raw output and use tools like pen and paper or an Excel spreadsheet when experimenting with parameter tuning. Finding the best approach depends on your preferences and needs. Eta, max_depth, and min_child_weight are indeed important parameters, but there are other valuable ones to consider.

Two parameters that can be particularly useful are ‘subsample‘ and ‘colsample_bytree.’ They have some similarities:

- ‘colsample_bytree‘:
  Similar to what we observe in Random Forest, this parameter controls how many features each tree gets to see at each iteration. The maximum value is 1.0. You can experiment with values like 0.3 and 0.6, and then fine-tune around those values.
- ‘subsample‘:
  Instead of sampling columns, this parameter allows you to sample rows. It means you can choose to provide only a percentage of the training data. For example, setting it to 0.5 means you randomly select 50% of the training data.

FINAL MODEL: CHOOSE BETWEEN DECISION TREE, RANDM FOREST, XGBOOST

Now, we revisit the best model of each type and evaluate their performance on the validation data. Based on these evaluations, we will select the overall best model and train it on the full training dataset. The final model will then be evaluated on the test set.

Best Decision Tree Model-

```
dt = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
dt.fit(X_train, y_train)

# Output:
# DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
```

Best Random Forest Model-

```
rf = RandomForestClassifier(n_estimators=200,
                            max_depth=10,
                            min_samples_leaf=3,
                            random_state=1)
rf.fit(X_train, y_train)

# Output:
# RandomForestClassifier(max_depth=10, min_samples_leaf=3, n_estimators=200
#                                                  random_state=1)
```

Best XGBoost Model-

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dtrain, num_boost_round=175)
```

Evaluating the AUC Score for each of the above on the Validation dataset

```
# Decision Tree
y_pred = dt.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)

# Output: 0.7850954203095104



# Random Forest
y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)

# Output: 0.8246258264512848



# XGBoost Model
y_pred = model.predict(dval)
roc_auc_score(y_val, y_pred)

# Output: 0.8309347073212081
```

we see XGBoost has the highest score. So, we'll train XGBoost model on Full Train Dataset, df_full_train

Extract the target variable, y_full_train and delete the status column from df_full_train to avoid accidental usage

```
y_full_train = (df_full_train.status == 'default').astype(int).values
y_full_train
# Output: array([0, 1, 0, ..., 0, 0, 1])
```

```
del df_full_train['status']
```

Create dictionaries for the DictVectorizer and then use the fit_transform method to obtain X_full_train. For X_test, we only need to call the transform method since the vectorizer has already been fitted.

```
dicts_full_train = df_full_train.to_dict(orient='records')

dv = DictVectorizer(sparse=False)
X_full_train = dv.fit_transform(dicts_full_train)

dicts_test = df_test.to_dict(orient='records')
X_test = dv.transform(dicts_test)
```

For XGBoost, create the DMatrix for Training using X_Full_train and Test using X_Test

```
feature_names = list(dv.get_feature_names_out())
dfulltrain = xgb.DMatrix(X_full_train, label=y_full_train,
                         feature_names=feature_names)

dtest = xgb.DMatrix(X_test, feature_names=feature_names)
```

Next, set the XGBoost Parameters, train, predict using test and evaluate AUC score

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 3,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',

    'nthread': 8,
    'seed': 1,
    'verbosity': 1,
}

model = xgb.train(xgb_params, dfulltrain, num_boost_round=175)
```

```
y_pred = model.predict(dtest)
roc_auc_score(y_test, y_pred)

# Output: 0.8289367577342261
```

Performance of final model on Test Dataset is close to that on Validation Dataset - we conclude that our model didn't overfit. The final model generalizes quite well on unseen data.

XGBoost models are often one of the best models at least for tabular data (dataframe with features). The downside of this is that **XGBoost** models are more complex, it's more difficult to tune, it has more parameters, and it's easier to overfit with **XGBoost**.