**End-To-End Automated Machine Learning Pipeline, Model Selection and Performance Evaluation**

**Performance Evaluation** is an indispensable process in any machine-learning project. Our machine learning process may not always result in an optimal model with the expected accuracy. Hence, performance measurement is needed here to evaluate the effectiveness of a trained model in prediction.

There are a great variety of metrics which are used in performance measurement but in general; they can be categorised based on the model type, 1) Classifier or 2) Regressor. In this article, we will only focus on the classifier type measurement by introducing seven common performance metrics used in a classification project. The seven metrics are as below:

However, there are other classification performance metrics out there that you will explore in this tutorial, you should look these up during your free time. While it may take a while to understand the underlying concept of some performance metrics above, the good news is that the implementation of those metrics has never been easier with [PyCaret](https://pycaret.gitbook.io/docs/).

PyCaret is an open-source, low-code machine learning library in Python that automates machine learning workflows. It is an end-to-end machine learning and model management tool that exponentially speeds up the experiment cycle and makes you more productive.

In the sections below, the concept of each performance metric will be explained by walking through a simple binary classification project based on the mobile network operator customer churn dataset. *The aim of this classification is to predict if a customer is going to churn to a different service provider.*

In comparison with the other open-source machine learning libraries, PyCaret is an alternate low-code library that can be used to replace hundreds of lines of code with few lines only. This makes experiments exponentially fast and efficient. PyCaret is essentially a Python wrapper around several machine learning libraries and frameworks such as scikit-learn, XGBoost, LightGBM, CatBoost, spaCy, Optuna, Hyperopt, Ray, and a few more. The design and simplicity of PyCaret are inspired by the emerging role of citizen data scientists. [**Citizen Data Scientists**](https://www.techtarget.com/searchbusinessanalytics/definition/citizen-data-scientist) are power users who can perform both simple and moderately sophisticated analytical tasks that would previously have required more technical expertise. A citizen data scientist is an individual who does some data science work for an organization but doesn't hold the title of data scientist or have a formal background in advanced analytics, statistics or related disciplines. Citizen data scientists can include business analysts, data-savvy business users, [business intelligence](https://www.techtarget.com/searchbusinessanalytics/definition/business-intelligence-BI) (BI) analysts and developers, data engineers and other workers.

**PyCaret workflow**

* **Getting Data:** How to import data from the PyCaret repository.
* **Setting up Environment:** How to set up a regression experiment in PyCaret and get started with building regression models.
* **Create Model:** How to create a model, perform cross-validation and evaluate regression metrics.
* **Tune Model:** How to automatically tune the hyperparameters of a regression model.
* **Plot Model:** How to analyze model performance using various plots.
* **Predict Model:** How to make predictions on new/unseen data.
* **Save / Load Model:** How to save/load a model for future use.

**Installing PyCaret Library**

Code cell:

# install pycaret

!pip install pycaret

Note: Some functionalities presented here may not be supported in an earlier used package of Scikit-learn.

**Case study (A) Churn and Retention Analysis for Mobile Network Customers**

**Step 1: Understanding The Business Problem**

The primary objective of the customer churn predictive model is to retain customers at the highest risk of churn by proactively engaging with them. For example: Offer a gift voucher or any promotional pricing and lock them in for an additional year or two to extend their lifetime value to the company.

We want a customer churn predictive model to predict the churn in advance (let’s say one month in advance, three months in advance, or even six months in advance — it all depends on the use-case). This means that you have to be extremely careful of the cut-off date i.e. You shouldn’t be using any information after the cut-off date as a feature in the machine learning model, otherwise it will be leakage. The period before the cut-off date is known as the Event.

Now you understand the business problem, let’s discuss how this machine-learning model will be used in the business. Read the below diagram from left to right (figure.1)

A diagram of a customer model

Description automatically generated

**Fig.1** Process flow for using ML for customer retention

* A model is trained on customer churn history (event period for X features and performance window for target variable).
* Every month active customer base is passed onto **Machine Learning Predictive Model** to return the probability of churn for each customer (in business lingo, this is sometimes called a score of churn).
* The list will be sorted from highest to lowest probability value (or score as they say it) and the customer retention teams will start engaging with the customer to stop the churn, normally by offering some kind of promotion or gift card to lock in few more years.
* Customers that have a very low probability of churn (or essentially model predicts no-churn) are happy customers. No actions are taken on them.

**Step 2: Data Loading**

For this tutorial, we are using a [Telecom Customer Churn](https://www.kaggle.com/blastchar/telco-customer-churn) dataset from Kaggle. The dataset already contains the target column that we can use as is.

#Import Data manipulation libraries

import pandas as pd

import numpy as np

# Expand truncated values in pandas data frame

pd.set\_option('display.max\_rows', 100)

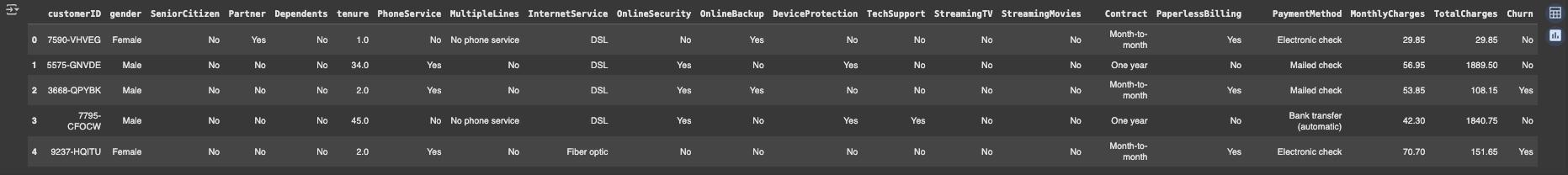
pd.set\_option('display.max\_columns', 100)

pd.set\_option('display.width', 100)

#Load your dataset

dataset = pd.read\_csv('/content/Telco-Customer-Churn2.csv')

dataset.head()



**Step 3: Data Understanding**

To investigate the dataset, check the dimensions of your dataset, the features data types and missing data. You will notice there are some blank spaces in some column.

#Check dataset dimensions

dataset.shape



# check data types

dataset.info()

# Investigate the portion of missing data values in each feature

dataset.isnull().sum()/len(dataset)\*100

A screenshot of a phone

Description automatically generated A screenshot of a computer

Description automatically generated

Intuitively, contract type, tenure (length of stay of the customer), and pricing plans typically provide very important information when it comes to customer churn or retention. Visualise the relationship in scatter plots:

# Visualise churn in the three types of contracts

import plotly.express as px

fig = px.scatter(x=dataset['tenure'], y=dataset['TotalCharges'],

color = dataset['Churn'], template = 'presentation',

opacity = 0.5, facet\_col = dataset['Contract'],

title = 'Customer Churn by Tenure, Charges, and Contract Type',

labels = {'x' : 'Customer Tenure', 'y' : 'Total Charges $'})

fig.show()

A graph showing a number of blue and orange dots

Description automatically generated

# Visualise churn in the three types of contracts

import plotly.express as px

fig = px.scatter(x=dataset['tenure'], y=dataset['MonthlyCharges'],

color = dataset['Churn'], template = 'presentation',

opacity = 0.5, facet\_col = dataset['Contract'],

title = 'Customer Churn by Tenure, Monthly Charges & Contract Type',

labels = {'x' : 'Customer Tenure', 'y' : 'MonthlyCharges $'})

fig.show()

A graph showing the amount of numbers and the amount of points

Description automatically generated with medium confidence

Notice that most churn can be seen in the contracts that are “Month-to-Month”. Also, I can see that as the tenure increases and so are the total charges. Why? Look at the monthly charges and their effect on customer churning. Think Business!

**Step 4: Data Preparation**

In order to demonstrate the use of the **predict\_model** function on unseen data, a sample of 10% of customers records has been withheld from the original dataset to be used for predictions. This should not be confused with a train/test split as this particular split is performed to simulate a real-life scenario. Another way to think about this is that these 10% records are not available at the time when this machine learning experiment was performed.

The remaining 90% of the data set will then be divided into 80:20 training and test split. The training data 80% will be folded with k=5 Cross Validation (CV) used to build many candidate machine learning models (Figure 2).

The models will be tested on the 20% test set to evaluate the each candidate model and produce performance metrics, one candidate model is selected (See Figure 2).

data = dataset.sample(frac=0.9, random\_state=786)

data\_unseen = dataset.drop(data.index)

data.reset\_index(drop=True, inplace=True)

data\_unseen.reset\_index(drop=True, inplace=True)

print('Data for Modeling: ' + str(data.shape))

print('Unseen Data For Predictions: ' + str(data\_unseen.shape))



A black background with white circles

Description automatically generated

**Step 5: Setting up PyCaret ML pipeline**

The setup function initializes the environment in PyCaret and creates the **transformation pipeline** to prepare the data for modelling and deployment. **setup** must be called before executing any other function in PyCaret. It takes two **mandatory parameters**: a **pandas data\_frame** and the **name of the target column**. All other parameters are optional and are used to customize the pre-processing pipeline. When setup is executed, PyCaret inference algorithm will automatically infer the data types for all features based on certain properties. The data type should be inferred correctly but this is not always the case. To account for this, PyCaret displays a table containing the features and their inferred data types after setup function is executed. If all of the data types are correctly identified enter can be pressed to continue or quit can be typed to end the experiment.  
*Ensuring that the data types are correct is really important in PyCaret as it automatically performs multiple type-specific preprocessing tasks which are imperative for machine learning models.* Alternatively, you can also use numeric\_features and categorical\_features parameters in the setup to pre-define the data types.

from pycaret.classification import \*

telecom = setup(data,target = 'Churn',

max\_encoding\_ohe = 100,

fold\_strategy = 'kfold',

fold = 5,

data\_split\_stratify = True ,

transformation = False,

train\_size = 0.8,

ignore\_features = ['customerID'],

ordinal\_features = {'Contract' : ['Month-to-month' ,'One year', 'Two year']},

normalize = True,

normalize\_method ='minmax',

low\_variance\_threshold = 0.1,

numeric\_imputation ='mean',

categorical\_imputation='mode',

session\_id = 43)

A screenshot of a computer

Description automatically generated

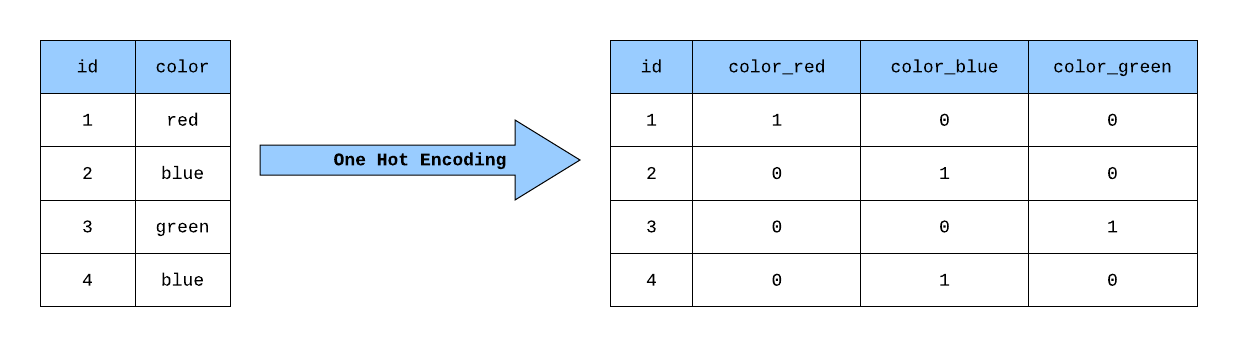
In the above code cell, the PyCaret **setup step**, Data preparation and cleaning pipeline is described. Unnecessary features can be ignored, while those with low variance (useless) can be dropped. Missing values are handled using simple imputation methods, mean for numeric features and the mode for categorical ones. Varied features magnitudes are handled with Normalisation. Both numeric and categorical features can be normalised in the setup step in data preprocessing. **Session\_id:** A pseudo-random number distributed as a seed in all functions for later reproducibility. If no session\_id is passed, a random number is automatically generated that is distributed to all functions. In this experiment, the session\_id is set as 123 for later reproducibility. **Original Data:** Displays the original shape of the dataset, the original shape of data (6339, 21) is transformed into (6339, 39). **Transformed Train Set:** Displays the shape of the transformed training set. Notice that the for the transformed train set. The number of features has increased from 19 to 39 due to categorical encoding.

We studied previously how to normalise features of numeric type using **min-max** and **standardization** scaling methods. In this experiment we specify the use **min-max** scaling for numeric features, with min-max scaling, the maximum value within a variable will be 1 while the minimum is transformed to 0, the values in between the minimum and maximum are transformed into ratios between 0 and 1.

A black background with a black square

Description automatically generated with medium confidence

Categorical features can be normalised by bringing all features to the same level of categories. **One-Hot-Encoding** can be used to normalise categorical features. For each categorical feature, each category is transformed into a feature with a value of 1 and 0 depending on its occurrence in each instance (see figure 3). Do not confuse **One-Hot-Encoding** with **Label Encoding. Label encoding** is used to transform categorical values from text format into numeric code format to maximize compatibility with a wider range of algorithms. Label Encoding does not create new variables (See figure 4). Label encoder is used in our experiment to transform the target feature.



**Fig.3** illustration of one-hot-encoding transformation

A screenshot of a computer

Description automatically generated

**Fig.4** Comparing one-hot-encoding to label encoding transformation

**Step 6: Examining the PyCaret ML pipeline**

Examining the preprocessing pipeline configuration can be important to ensure that your dataset is prepared adequately. You can use the **get\_config( )** to perform your checks.

# To get a list of all possible checks on configurations

get\_config()

A screen shot of a computer program

Description automatically generated

# to view the training set X inputs before transformation

get\_config('X\_train')

A screenshot of a black and white screen

Description automatically generated

# To view the training set X inputs after transformation

get\_config('X\_train\_transformed')

A screenshot of a black and white table

Description automatically generated

# To view the training set Y target before transformation

get\_config('y\_train')

A screenshot of a phone

Description automatically generated

# To view the training set Y target after transformation

get\_config('y\_train\_transformed')

A screenshot of a cell phone

Description automatically generated

**Step 7: Building candidate ML classification models**

Comparing all candidate models to evaluate performance is the recommended starting point for modelling once the setup is completed (unless you exactly know what kind of model you need, which is often not the case). This function trains all models in the model library and scores them using k-fold cross-validation for metric evaluation. The output prints a scoring grid that shows average Accuracy, AUC, Recall, Precision, F1-Score, MCC and Kappa across the folds (K = 5) along with the training time each model took.

# compare all models

best\_model = compare\_models(sort = 'Accuracy')

A screenshot of a graph

Description automatically generated

One line of code and we have trained and evaluated over 15 models using cross-validation. The scoring grid printed above highlights the highest performing metric for comparison purposes only. The grid by default is sorted using Accuracy which can be changed by passing sort parameter. For example, **compare\_models(sort = 'RMSLE')** will sort the grid by RMSLE (lower to higher since lower is better). By default, the **compare\_models** return the best performing model based on default sort order but can be used to return a list of top N models by using **n\_select** parameter.

There are additional classifiers that you can build in PyCaret, those are non-turbo algorithms which take longer time to build. By default the compare\_models build Turbo classifiers. To see the full list of classifiers use **models()**, to build a specific classifier use **create\_model** function

#To view all available algorithms

models()

A screenshot of a computer

Description automatically generated

**create\_model** is the most granular function in PyCaret and is often the foundation behind most of the PyCaret functionalities. As the name suggests this function trains and evaluates a model using cross-validation that can be set with the fold parameter. The output prints a scoring grid that shows the performance metrics by fold. For the remaining part of this tutorial, let’s build a **KNN model** for example with k=5 and using the Manhattan distance.

# to create a KNN model with k=5

knn=create\_model('knn',n\_neighbors = 5 , metric = 'manhattan')

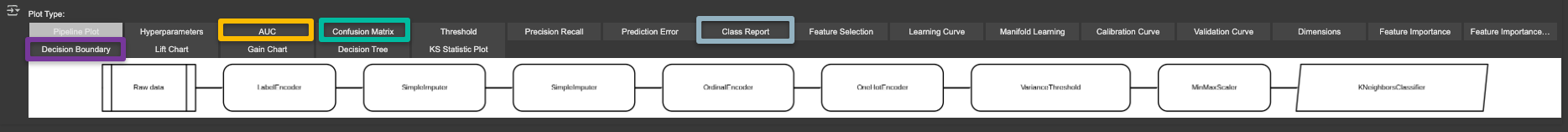
A screenshot of a graph

Description automatically generated

One way to analyze the performance of models is to use the **evaluate\_model** function which displays a user interface for all of the available plots for a given model. It internally uses the **plot\_model** function.

#Graphical plot

evaluate\_model(knn)



A screenshot of a computer

Description automatically generated

A screenshot of a graph

Description automatically generated

A screen shot of a graph

Description automatically generated

A screenshot of a computer

Description automatically generated

You can now compare the KNN model performance to your **best\_model**, Gradient Boosting Classifier

#Graphical plot

evaluate\_model(best\_model)



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a graph

Description automatically generated

A screenshot of a computer

Description automatically generated

**Step 8: Hyperparameter Tuning of candidate ML classification models**

When a model is created using the **create\_model** function it uses the **default hyperparameters** to train the model. In order to tune hyperparameters, the **tune\_model** function is used. This function automatically tunes the hyperparameters of a model using **RandomGridSearch**on a pre-defined search space. The output prints a scoring grid that shows classification performance metrics by fold. To use the custom search grid, you can pass **custom\_grid** parameter in the **tune\_model** function. Let’s tune the KNN model and observe any improvement in the performance metrics.

# Tune the KNN model

tuned\_knn = tune\_model(knn)

A screenshot of a graph

Description automatically generated

When comparing both the KNN model before and after tuning, we notice an improvement across all performance metrics. You can also view the KNN hyperparameters before and after tuning using **print( )** function.

# To view the knn hyperparameters before tuning

print(knn)



# To view the knn hyperparameters after tuning

print(tuned\_knn)



Use the **evaluate\_model( )** function to assess the performance metrics for the **tuned KNN model** by examining the graphical plots including, the confusion matrix and the classification report.

#Graphical plot of tuned KNN

evaluate\_model(tuned\_knn)

A screenshot of a computer

Description automatically generated

A screenshot of a graph

Description automatically generated

A screen shot of a graph

Description automatically generated

Let’s tune the gbc model (best\_model) and observe any improvement in the performance metrics.

#Tune the Gradient Boosting Classifer

tuned\_best\_model=tune\_model(best\_model)

A screenshot of a computer

Description automatically generated

Observe that the tunning of the best model hyperparameters has resulted in a slight deterioration of classification accuracy performance compared to the original gbc model. Therefore. We will stick to our best model performance metrics.

**Step 9: Model selection with Dollar ($) Impact analysis**

In a churn model, often, the reward of *true positives* is way different from the cost of *false positives*. Let’s use the following assumptions:

* *£1,000 discount will be offered on the term of the agreement to all the customers identified as churn positive (True Positive + False Positive)*
* *If we are able to stop a customer leaving (churn), we will gain £5,000 in customer lifetime value.*

Using these assumptions and the confusion matrix above, we can calculate the $ impact for our **best\_model** and our **tuned\_knn** model.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| **Gradient Boosting Classifier** | Customers | Value | Total |
| Ture Positive (Correct churn) | 172 | $5000 (Profit) | $860,000 |
| True Positive + False Positive | 172 + 78 | $1000 (Discount Loss) | $250,000 |
|  | | **Gross Profit** | $610,000 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Tuned KNN Classifier** | Customers | Value | Total |
| Ture Positive (Correct churn) | 184 | $5000 (Profit) | $920,000 |
| True Positive + False Positive | 184 + 118 | $1000 (Discount Loss) | $302,000 |
|  | | **Gross Profit** | $618,000 |

We can see that the tuned KNN model has generated higher Gross Profit by $8,000. However, this additional $8K required spending an additional $52,000 discounts above the discount value spent in the Gradient Boosting Classifier model. The business will have to decide on which model to go for based on their budget. So, in this case they decided to go with the cheaper model, our **best\_model**, gbc.

**Step 10: Predictions on the test data**

Before finalizing the model, it is advisable to perform one final check by predicting the test set and reviewing the evaluation metrics. If you look at the information in step 4 above, you will see that 20% (1268 samples) of the data has been separated out as a test sample. All of the evaluation metrics we have seen above are cross-validated results based on the training set (80%) only.

Now, using our selected final model stored in the **best\_model** , we will predict the test sample and evaluate the metrics to see if they are materially different than the CV results.

# Perform a direct test of the best\_model on the test set

predict\_model(best\_model)

A screenshot of a computer

Description automatically generated

The accuracy on the test set is **0.8076** compared to **0.8063** achieved on the best\_model CV results (in step 7). This is not a significant difference. If there is a large variation between the test and CV results, then this would normally indicate over-fitting but could also be due to several other factors and would require further investigation. In this case, we will move forward with finalizing the model and predicting on unseen data (the 10% that we had separated in the beginning and never exposed to PyCaret).

**Step 11: Finalize selected model for deployment**

**Model finalization** is the last step in the experiment. A machine learning workflow in PyCaret starts with the setup, followed by comparing all models using compare\_models and shortlisting a few candidate models (based on the metric of interest) to perform several modelling techniques such as hyperparameter tuning, ensembling, stacking, etc. This workflow will eventually lead you to the **best model** for use in making predictions on **new and** **unseen data**.  
The **finalize\_model** function fits the model onto **the complete dataset** including the test set sample (20% in this case). The purpose of this function is to **train the model on the complete dataset before it is deployed in production** (See figure 5).

A black and white image of a server

Description automatically generated

**Fig.5** Illustration of finalising the selected best model for deployment

final\_best\_model = finalize\_model(best\_model)

print(final\_best\_model)

A computer screen shot of white text

Description automatically generated

**Warning!**   
One final word of caution. Once the model is finalized using the **finalize\_model**, the entire dataset including the test set is used for training. As such, if the model is used for predictions on the test set after **finalize\_model** is used, the information grid performance printed will be misleading as you are trying to predict on the same data that was used for modelling. In order to demonstrate this point only, we will use **final\_best\_model** under **predict\_model** function to compare the information grid with the one above in section 7.

predict\_model(final\_best\_model)

A screenshot of a computer

Description automatically generated

Notice how the accuracy in the **final\_best\_model** has increased to **0.8289** from **0.8076** , and the AUC increased from **0.8590 to 0.8908** even though the model is the same. This is because the**final\_best\_model** variable is trained on the complete dataset including the test set.

**Step 12: Predict on the unseen dataset**

The **predict\_model** function is used to predict the unseen/new dataset. The only difference from the section above is that this time we will pass the **data\_unseen** parameter. **data\_unseen** is the variable created at the beginning of the tutorial and contains 10% (704 samples) of the original dataset which was never exposed to PyCaret. (see step 4 for explanation).

#Predicting on unseen data

unseen\_predictions = predict\_model(final\_best\_model, data=data\_unseen)

unseen\_predictions.head()

A screenshot of a computer

Description automatically generated

Save the predictions on the unseen data as a .csv file and download it from Google Colab.

#This will save the dataset without the raws indeces

unseen\_predictions.to\_csv(r'/content/unseen\_prediction.csv', index=True)

A screenshot of a computer

Description automatically generated

**Step 13: Save your final model**

We have now finished the experiment by finalizing the **best\_model** which is now stored in **final\_best\_model** variable. We have also used the model stored in **final\_best\_model** to predict **data\_unseen**. This brings us to the end of our experiment, but one question is still to be asked: What happens when you have more new data to predict? Do you have to go through the entire experiment again? The answer is no, PyCaret's inbuilt function **save\_model** allows you to save the model along with the entire transformation pipeline for later use.

save\_model(final\_best\_model,'Final\_best\_Model(gbc)')

A screenshot of a computer program

Description automatically generated

**Step 14: Loading your saved model**

To load a saved model at a future date in the same or an alternative environment, upload your saved model file to your Google Colab environment. Use PyCaret’s **load\_model**function to easily apply the saved model on new unseen data for prediction.

import pycaret

from pycaret.classification import \*

saved\_final\_best\_model= load\_model('/content/Final\_best\_Model(gbc)')

A screenshot of a computer program

Description automatically generated

Transformation Pipeline and Model Successfully Loaded. Once the model is loaded in the environment, you can simply use it to predict any new data using the same predict\_model function. Below we have applied the loaded model to predict new customers data which are in the **New\_production\_Data.csv** that came from production. The production data does not have labels, so our model will predict if these fresh customers are going to churn or not so they can be contacted for retention and offered a discount. Upload your production data to Google Colab and load your file.

#Load your production dataset

import pandas as pd

new\_production\_data = pd.read\_csv('/content/New\_production\_Data.csv')

new\_production\_data.head()



new\_predictions = predict\_model(saved\_final\_best\_model, data = new\_production\_data)

new\_predictions.head()

A screenshot of a computer

Description automatically generated

**HOMEWORK HOMEWORK HOMEWORK HOMEWORK**

**Case study (B) Predicting Diamonds Prices**

**Step 1: Understanding The Business Problem**

we will use the dataset from PyCaret’s dataset repository. The data contains 6000 records for training. Short descriptions of each column are as follows:

* **ID:** Uniquely identifies each observation (diamond)
* **Carat Weight:** The weight of the diamond in metric carats. One carat is equal to 0.2 grams, roughly the same weight as a paperclip
* **Cut:** One of five values indicating the cut of the diamond in the following order of desirability (Signature-Ideal, Ideal, Very Good, Good, Fair)
* **Color:** One of six values indicating the diamond’s color in the following order of desirability (D, E, F — Colorless, G, H, I — Near-colorless)
* **Clarity:** One of seven values indicating the diamond’s clarity in the following order of desirability (F — Flawless, IF — Internally Flawless, VVS1 or VVS2 — Very, Very Slightly Included, or VS1 or VS2 — Very Slightly Included, SI1 — Slightly Included)
* **Polish:** One of four values indicating the diamond’s polish (ID — Ideal, EX — Excellent, VG — Very Good, G — Good)
* **Symmetry:** One of four values indicating the diamond’s symmetry (ID — Ideal, EX — Excellent, VG — Very Good, G — Good)
* **Report:** One of two values “AGSL” or “GIA” indicates which grading agency reported the qualities of the diamond qualities
* **Price:** The amount in USD that the diamond is valued **Target Column**

**Step 2: Data Loading**

from pycaret.datasets import get\_data

dataset = get\_data('diamond')

**Step 3: Data Understanding**

#check the dimensions of data

dataset.shape

# check data types

dataset.info()

# Investigate the portion of missing data values in each feature

dataset.isnull().sum()/len(dataset)\*100

**Step 4: Data Preparation**

In order to demonstrate the use of the **predict\_model** function on unseen data, a sample of 600 records has been withheld from the original dataset to be used for predictions. This should not be confused with a train/test split as this particular split is performed to simulate a real-life scenario. Another way to think about this is that these 600 records are not available at the time when this machine learning experiment was performed.

data = dataset.sample(frac=0.9, random\_state=876)

data\_unseen = dataset.drop(data.index)

data.reset\_index(drop=True, inplace=True)

data\_unseen.reset\_index(drop=True, inplace=True)

print('Data for Modeling: ' + str(data.shape))

print('Unseen Data For Predictions: ' + str(data\_unseen.shape))

**Step 5: Setting up PyCaret ML pipeline**

from pycaret.regression import \*

s = setup(data = data, target = 'Price', session\_id=123)

**Step 6: Building candidate ML classification models**

Comparing all models to evaluate performance is the recommended starting point for modeling once the setup is completed (unless you exactly know what kind of model you need, which is often not the case). This function trains all models in the model library and scores them using k-fold cross-validation for metric evaluation. The output prints a scoring grid that shows average MAE, MSE, RMSE, R2, RMSLE, and MAPE across the folds (10 by default) along with the training time each model took.

best = compare\_models(sort='R2')

One line of code and we have trained and evaluated over 20 models using cross-validation. The scoring grid printed above highlights the highest performing metric for comparison purposes only. The grid by default is sorted using R2 (highest to lowest) which can be changed by passing sort parameter. For example, **compare\_models(sort = 'RMSE')** will sort the grid by RMSLE (lower to higher since lower is better). If you want to change the fold parameter from the default value 10 to a different value then you can use the fold parameter. For example **compare\_models(fold = 5)** will compare all models on 5 fold cross-validation. Reducing the number of folds will improve the training time. By default, the compare\_models return the best performing model based on default sort order but can be used to return a list of top N models by using n\_select parameter.

**create\_model** is the most granular function in PyCaret and is often the foundation behind most of the PyCaret functionalities. As the name suggests this function trains and evaluates a model using cross-validation that can be set with the fold parameter. The output prints a scoring grid that shows MAE, MSE, RMSE, R2, RMSLE, and MAPE by fold. For the remaining part of this tutorial, we will work with LR and KNN models as our candidate models. The selections are for illustration purposes only and do not necessarily mean they are the top-performing or ideal for this type of data.

There are 25 regressors available in the model library of PyCaret. To see a complete list of all regressors either check the docstring or use models function to see the library.

models()

**Creating a K-Nearest Neighbours regression model**

knn = create\_model('knn',n\_neighbors=3,fold = 10)

print(knn)

**Creating a Multiple Linear Regression model**

lr=create\_model('lr',fold = 10)

print(lr)

Notice that the Mean score of all models matches with the score printed in compare\_models. This is because the metrics printed in the compare\_models score grid are the average scores across all CV folds. Similar to the compare\_models, if you want to change the fold parameter from the default value of 10 to a different value then you can use the fold parameter in the create\_model function. For Example: create\_model('dt', fold = 5) to create a Decision Tree using 5 fold cross-validation.

**Step 7: Hyperparameter Tuning of candidate ML regression models with custom parameter grid**

When a model is created using the create\_model function it uses the default hyperparameters to train the model. In order to tune hyperparameters, the tune\_model function is used. This function automatically tunes the hyperparameters of a model using RandomGridSearch on a pre-defined search space. The output prints a scoring grid that shows MAE, MSE, RMSE, R2, RMSLE, and MAPE by fold. To use the custom search grid, you can pass custom\_grid parameter in the tune\_model function.

tuned\_knn = tune\_model(knn)

print(tuned\_knn)

import numpy as np

custom\_knn\_grid = {'n\_neighbors': (1,3,5,7,9),'weights': ('uniform', 'distance')}

custom\_tuned\_knn = tune\_model(knn, custom\_grid = custom\_knn\_grid, fold=10, optimize = 'MAE')

print(custom\_tuned\_knn)

By default, tune\_model optimizes R2 but this can be changed using THE optimize parameter. For example, tune\_model(dt, optimize = 'MAE')will search for the hyperparameters of a Decision Tree Regressor that results in the lowest MAE instead of highest R2. For the purposes of this example, we have used the default metric R2 for the sake of simplicity only. The methodology behind selecting the right metric to evaluate a regressor is beyond the scope of this tutorial.  
Metrics alone are not the only criteria you should consider when finalizing the best model for production. Other factors to consider include training time, the standard deviation of k-folds, etc. For now, let’s move forward considering the Tuned KNN Model stored in the tuned\_knn variable as our best model for the remainder of this tutorial.

**Step 7: Plot a model**

Before model finalization, the plot\_model function can be used to analyze the performance across different aspects such as Residuals Plot, Prediction Error, Feature Importance, etc. This function takes a trained model object and returns a plot based on the test / hold-out set. There are over 10 plots available, please see the plot\_model documentation for the list of available plots.

**Residual Plot**

plot\_model(knn)

plot\_model(tuned\_knn)

**Prediction Error Plot**

plot\_model(knn, plot = 'error')

plot\_model(tuned\_knn, plot = 'error')

Another way to analyze the performance of models is to use the evaluate\_model function which displays a user interface for all of the available plots for a given model. It internally uses the **plot\_model** function.

evaluate\_model(tuned\_knn)

**Step 8: Predict on Test sample**

Before finalizing the model, it is advisable to perform one final check by predicting the test/hold-out set and reviewing the evaluation metrics. If you look at the information grid in step 5 above, you will see that 30% (1621 samples) of the data has been separated out as a test/hold-out sample. All of the evaluation metrics we have seen above are cross-validated results based on the training set (70%) only. Now, using our final trained model stored in the tuned\_knn , we will predict the hold-out sample and evaluate the metrics to see if they are materially different than the CV results.

predict\_model(tuned\_knn);

The R2 on the test/hold-out set is 0.8266 compared to 0.8129 achieved on tuned\_knn CV results (in section 10.2 above). This is not a significant difference. If there is a large variation between the test/hold-out and CV results, then this would normally indicate over-fitting but could also be due to several other factors and would require further investigation. In this case, we will move forward with finalizing the model and predicting on unseen data (the 10% that we had separated in the beginning and never exposed to PyCaret).

**Step 9: Finalize Model for Deployment**

Model finalization is the last step in the experiment. A machine learning workflow in PyCaret starts with the setup, followed by comparing all models using compare\_models and shortlisting a few candidate models (based on the metric of interest) to perform several modeling techniques such as hyperparameter tuning, ensembling, stacking, etc. This workflow will eventually lead you to the best model for use in making predictions on new and unseen data.  
The finalize\_model function fits the model onto the complete dataset including the test/hold-out sample (30% in this case). The purpose of this function is to train the model on the complete dataset before it is deployed in production.

final\_knn = finalize\_model(tuned\_knn)

print(final\_knn)

**Warning!**   
One final word of caution. Once the model is finalized using the finalize\_model, the entire dataset including the test/hold-out set is used for training. As such, if the model is used for predictions on the hold-out set after finalize\_model is used, the information grid printed will be misleading as you are trying to predict on the same data that was used for modeling.

In order to demonstrate this point only, we will use final\_knn under predict\_model to compare the information grid with the one above in step 6.

predict\_model(final\_knn)

Notice how the R2 in the final\_knn has increased to **0.9991 from 0.8266** , even though the model is the same. This is because the final\_knn variable is trained on the complete dataset including the test/hold-out set.

**Step 10: Predict on Unseen Data**

The predict\_model function is used to predict the unseen/new dataset. The only difference from the section above is that this time we will pass the data\_unseen parameter. data\_unseen is the variable created at the beginning of the tutorial and contains 10% (600 samples) of the original dataset which was never exposed to PyCaret. (see section 5 for explanation)

unseen\_predictions = predict\_model(final\_knn, data=data\_unseen)

unseen\_predictions.head()

The Label column is added onto the data\_unseen set. The label is the predicted value using the final\_knn model. If you want predictions to be rounded, you can use the round parameter inside the predict\_model. You can also check the metrics on this since you have an actual target column Price available. To do that we will use the pycaret.utils module.

**Step 11: Saving the Model**

We have now finished the experiment by finalizing the tuned\_knn model which is now stored in final\_knn variable. We have also used the model stored in final\_knn to predict data\_unseen. This brings us to the end of our experiment, but one question is still to be asked: What happens when you have more new data to predict? Do you have to go through the entire experiment again? The answer is no, PyCaret's inbuilt function save\_model allows you to save the model along with the entire transformation pipeline for later use.

save\_model(final\_knn,'Final knn Model 29052024')

**Step 12: Loading the saved model**

To load a saved model at a future date in the same or an alternative environment, we would use PyCaret’s load\_model function and then easily apply the saved model on new unseen data for prediction.

saved\_final\_knn = load\_model('Final knn Model 29052024')

Once the model is loaded in the environment, you can simply use it to predict any new data using the same predict\_model function. Below we have applied the loaded model to predict the same data\_unseen that we used in section 10 above.

new\_prediction = predict\_model(saved\_final\_knn, data=data\_unseen)

new\_prediction.head()

Notice that the results of unseen\_predictions and new\_prediction are identical.