What is PyCaret?

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.

Compared 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 a 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, a term first used by Gartner. Citizen Data Scientists are power users who can perform both simple and moderately sophisticated analytical tasks that would previously have required more technical expertise.

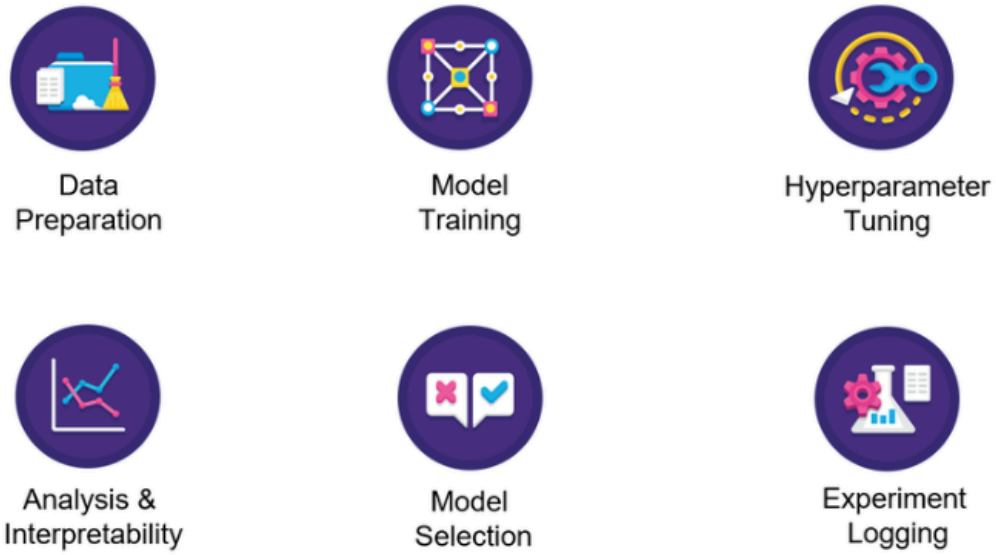
PyCaret is an open-source, low-code machine learning library in Python that aims to reduce the hypothesis to insight cycle time in an ML experiment. It enables data scientists to perform end-to-end experiments quickly and efficiently. In comparison with the other open-source machine learning libraries, PyCaret is an alternate low-code library that can be used to perform complex machine learning tasks with only a few lines of code. PyCaret is simple and easy to use.

PyCaret deployment capabilities

PyCaret is a deployment ready library in Python which means all the steps performed in an ML experiment can be reproduced using a pipeline that is reproducible and guaranteed for production. A pipeline can be saved in a binary file format that is transferable across environments.

PyCaret is seamlessly integrated with BI

PyCaret and its Machine Learning capabilities are seamlessly integrated with environments supporting Python such as Microsoft Power BI, Tableau, Alteryx, and KNIME to name a few. This gives immense power to users of these BI platforms who can now integrate PyCaret into their existing workflows and add a layer of Machine Learning with ease.



Installation of PyCaret

You can install PyCaret with Python's pip package manager:

```
pip install pycaret
```

In [36]:
```
# import sys
# !{sys.executable} -m pip install pycaret
```

In [2]:
```
print(sys.executable)
```

```
C:\Users\User\AppData\Local\Programs\Python\Python311\python.exe
```

Classification Problem

PyCaret's Classification Module is a supervised machine learning module that is used for classifying elements into groups.

The goal is to predict the categorical class labels which are discrete and unordered. Some common use cases include predicting customer default (Yes or No), predicting customer churn (customer will leave or stay), the disease found (positive or negative).

This module can be used for binary or multiclass problems.

First we will load a sample dataset from PyCaret itself but we can use any data and load it using pandas then read it as csv like how we normally do

In [7]:
```
from pycaret.datasets import get_data
data = get_data("diabetes")
```

	Number of times pregnant	Plasma glucose concentration a 2 hours in an oral glucose tolerance test	Diastolic blood pressure (mm Hg)	Triceps skin fold thickness (mm)	2-Hour serum insulin (mu U/ml)	Body mass index (weight in kg/(height in m)^2)	D pe fu
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

setup()

Then we will Setup and experiment,

`setup()` function initializes the training environment and creates the transformation pipeline. Setup function must be called before executing any other function. It takes two required parameters: `data` and `target`. All the other parameters are optional.

In PyCaret, the setup function is a crucial step in preparing your dataset for machine learning tasks. It performs a variety of preprocessing tasks automatically.

There are a number of additional parametersthat can be set for the experiment within the function

In [10]:
```
from pycaret.classification import *
s = setup(data, target = 'Class variable', session_id = 123)
```

	Description	Value
0	Session id	123
1	Target	Class variable
2	Target type	Binary
3	Original data shape	(768, 9)
4	Transformed data shape	(768, 9)
5	Transformed train set shape	(537, 9)
6	Transformed test set shape	(231, 9)
7	Numeric features	8
8	Preprocess	True
9	Imputation type	simple
10	Numeric imputation	mean
11	Categorical imputation	mode
12	Fold Generator	StratifiedKFold
13	Fold Number	10
14	CPU Jobs	-1
15	Use GPU	False
16	Log Experiment	False
17	Experiment Name	clf-default-name
18	USI	e6d9

You can see above are all the different parameters we could have used, since we didnt, it used the default, we can see there are parameters such as:

- `data` (required) The dataset you want to work with, provided as a Pandas DataFrame.
- `target` (required) The name of the target variable (the column you're trying to predict).
- `train_size` Fraction of data to be used for training. Default is 0.7, meaning 70% training and 30% test.
- `test_data` If you already have a separate test set, you can provide it here instead of splitting from the training data.
- `session_id` Seed for random number generation to ensure reproducibility. If you set this, the same random processes will yield identical results on different runs.
- `fold_strategy` defines the cross-validation strategy. Common options are:

  ```
  'kfold' (default)
  'stratifiedkfold'
  'groupkfold'
  'timeseries', etc.
  ```
- `fold` Number of folds for cross-validation. Default is 10.
- `fold_shuffle` Whether to shuffle data before splitting into folds. Default is True.
- `index` Column to use as an index.
- `ignore_features` List of features to ignore during modeling.
- `categorical_features` List of categorical columns that should be treated as categorical.
- `numeric_features` List of numeric columns to be treated as numeric.
- `date_features` List of columns to be treated as date variables.
- `categorical_imputation` The method to impute missing values in categorical columns. Default is 'mode'.

- `numeric_imputation` The method to impute missing values in numeric columns. Default is 'mean'.

- `normalize` Whether to scale the numeric data (True/False). Default is False.

- `normalize_method` Method to use for normalization:

  ```
  'zscore' (default)
  'minmax'
  'maxabs'
  'robust'.
  ```

- `transformation` Whether to apply transformations like power transform or log transform to make data more Gaussian-like. Default is False.

- `transformation_method` Method for transformation:

  ```
  'yeo-johnson' (default)
  'quantile'.
  ```

- `handle_unknown_categorical` Whether to handle unseen categories in categorical features. Default is True.

- `unknown_categorical_method` Method to handle unseen categories in categorical features:

  ```
  'least_frequent'
  'most_frequent'.
  ```

- `pca` Whether to apply Principal Component Analysis (PCA) for dimensionality reduction. Default is False.

- `pca_method` Method of PCA:

  ```
  'linear' (default)
  'kernel'
  'incremental'.
  ```

- `pca_components` Number of principal components to keep.

- `feature_selection` Whether to apply feature selection to the data. Default is False.

- `feature_selection_threshold` Threshold to drop features based on importance. Default is 0.8.

- `feature_interaction` Whether to automatically create interaction features. Default is False.
- `feature_ratio` Whether to automatically create polynomial features (ratios between features). Default is False.
- `polynomial_features` Whether to generate polynomial features. Default is False.
- `polynomial_degree` The degree for polynomial features. Default is 2.
- `remove_outliers` Whether to remove outliers from data. Default is False.
- `outliers_threshold` The threshold for identifying outliers. Default is 0.05.
- `combine_rare_levels` Whether to group rare categories in categorical features into one category. Default is False.
- `rare_level_threshold` Threshold below which categories are considered rare. Default is 0.1.
- `bin_numeric_features` List of numeric features to be binarized (converted into categorical by binning).
- `remove_multicollinearity` Whether to remove highly correlated features. Default is False.
- `multicollinearity_threshold` Threshold for correlation above which features are considered multicollinear. Default is 0.9.
- `create_clusters` Whether to create cluster labels as a new feature. Default is False.
- `cluster_iter` Number of iterations for clustering. Default is 20.
- `polynomial_threshold` Threshold for dropping polynomial features based on importance. Default is 0.5.

- `group_features` List of features to group (useful for creating meta-features).
- `group_names` Names for feature groups if `group_features` is provided.
- `use_gpu` If True, PyCaret will attempt to use GPU acceleration (with algorithms that support it).
- `profile` Whether to generate an EDA report of the dataset using `pandas_profiling`. Default is False.
- `log_experiment` Whether to log the entire experiment for easy reproducibility. Default is False.
- `log_plots` Log plots of the training process, like confusion matrix, feature importance, etc. Default is False.
- `silent` If True, disables all prompts and messages during the setup process.

These parameters give you a lot of flexibility to customize how your data is prepared before training machine learning models. You can tune everything from feature engineering to handling missing values and scaling, making setup() a powerful tool for automating many common preprocessing tasks.

Train and Evaluate Models

Once setting up we can do Training, we can use the `compare_models()` method.

The `compare_models()` function in PyCaret is a powerful utility that automatically trains and evaluates multiple machine learning models on your dataset. It ranks them based on a performance metric, allowing you to quickly compare their effectiveness and select the best-performing model.

It trains various machine learning models available in PyCaret (like Logistic Regression, Random Forest, XGBoost, etc.) on your data.

Models are evaluated using cross-validation (the number of folds is specified by the fold parameter in setup()).

After training, it ranks the models based on a specified evaluation metric (e.g., accuracy, AUC, F1-score).

By default, the models are ranked by accuracy in classification tasks or R-squared in regression tasks, but you can change this with the sort parameter.

The function returns the best-performing model based on the ranking so we can store it in a variable

In [11]:
```
best = compare_models()
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa
lr	Logistic Regression	0.7689	0.8047	0.5602	0.7208	0.6279	0.4641
ridge	Ridge Classifier	0.7670	0.8060	0.5497	0.7235	0.6221	0.4581
lda	Linear Discriminant Analysis	0.7670	0.8055	0.5550	0.7202	0.6243	0.4594
rf	Random Forest Classifier	0.7485	0.7911	0.5284	0.6811	0.5924	0.4150
nb	Naive Bayes	0.7427	0.7955	0.5702	0.6543	0.6043	0.4156
gbc	Gradient Boosting Classifier	0.7373	0.7914	0.5550	0.6445	0.5931	0.4013
ada	Ada Boost Classifier	0.7372	0.7799	0.5275	0.6585	0.5796	0.3926
et	Extra Trees Classifier	0.7299	0.7788	0.4965	0.6516	0.5596	0.3706
qda	Quadratic Discriminant Analysis	0.7282	0.7894	0.5281	0.6558	0.5736	0.3785
lightgbm	Light Gradient Boosting Machine	0.7133	0.7645	0.5398	0.6036	0.5650	0.3534
knn	K Neighbors Classifier	0.7001	0.7164	0.5020	0.5982	0.5413	0.3209
dt	Decision Tree Classifier	0.6928	0.6512	0.5137	0.5636	0.5328	0.3070
xgboost	Extreme Gradient Boosting	0.6891	0.7572	0.5292	0.5668	0.5438	0.3089
dummy	Dummy Classifier	0.6518	0.5000	0.0000	0.0000	0.0000	0.0000

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa
svm	SVM - Linear Kernel	0.5954	0.5914	0.3395	0.4090	0.2671	0.0720

Now we will have the best model stored inside the `best` variable, we can simply print it and it will give us all the parameters it used

In [13]:
```
print(best)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_inte
rcept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1
000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=123, solver='lbfgs', tol=0.0001,
verbose=0,
                   warm_start=False)
```

We could also pass some Parameters in the `compare_models()` method such as:

- `fold` : The number of cross-validation folds. Default is 10.

`sort` : Metric to sort the models by. Default for classification is accuracy. `n_select` : Selects the top N models instead of just one. If you set n_select=3, it will return the top 3 models.

- `include` : A list of models to include in the comparison. Example: include=['lr', 'rf'] for logistic regression and random forest.
- `exclude` : A list of models to exclude from the comparison.
- `turbo` : When set to True, it runs a smaller subset of models for faster comparison. Default is True.

Example of a more customized usage:

```
    best_model = compare_models(fold=5,
sort='F1', include=['lr', 'rf', 'xgboost'])
```

You can either use `compare_models()` or `create_model()` based on your goal

1. `compare_models()`:

- Purpose: If you want to quickly compare multiple models to see which one performs the best based on a specific evaluation metric (like accuracy, F1, or AUC), this is the way to go.
- How it works: It trains multiple models automatically, ranks them based on performance, and returns the best model (or the top N models if you use n_select).
- Best for: When you are unsure which model might work best for your dataset and want an overall comparison.

2. `create_model()`:

- Purpose: When you already know which model you want to use and just want to train it with cross-validation, this is the function to use.
- How it works: You specify the exact model (like 'rf' for Random Forest or 'lr' for Logistic Regression), and it will train the model and evaluate it with cross-validation.
- Best for: When you have a specific model in mind that you want to train and evaluate
- Example:

```
rf_model = create_model('rf')
```

evaluate_model()

The `evaluate_model()` function in PyCaret is designed to visualize and assess the performance of a trained machine learning model using a series of plots. It provides a comprehensive view of various performance metrics, allowing you to understand how well the model is likely to perform on unseen data.

It generates a variety of plots to visualize the model's performance, including: Confusion Matrix, ROC Curve, Precision-Recall Curve, Feature Importance, Residuals Plot (for regression models)

This function allows you to evaluate the model based on different metrics and visualize how well it classifies or predicts outcomes, which

can help identify strengths and weaknesses.

The plots are based on cross-validation results, ensuring that the evaluation is robust and not biased by any particular train-test split.

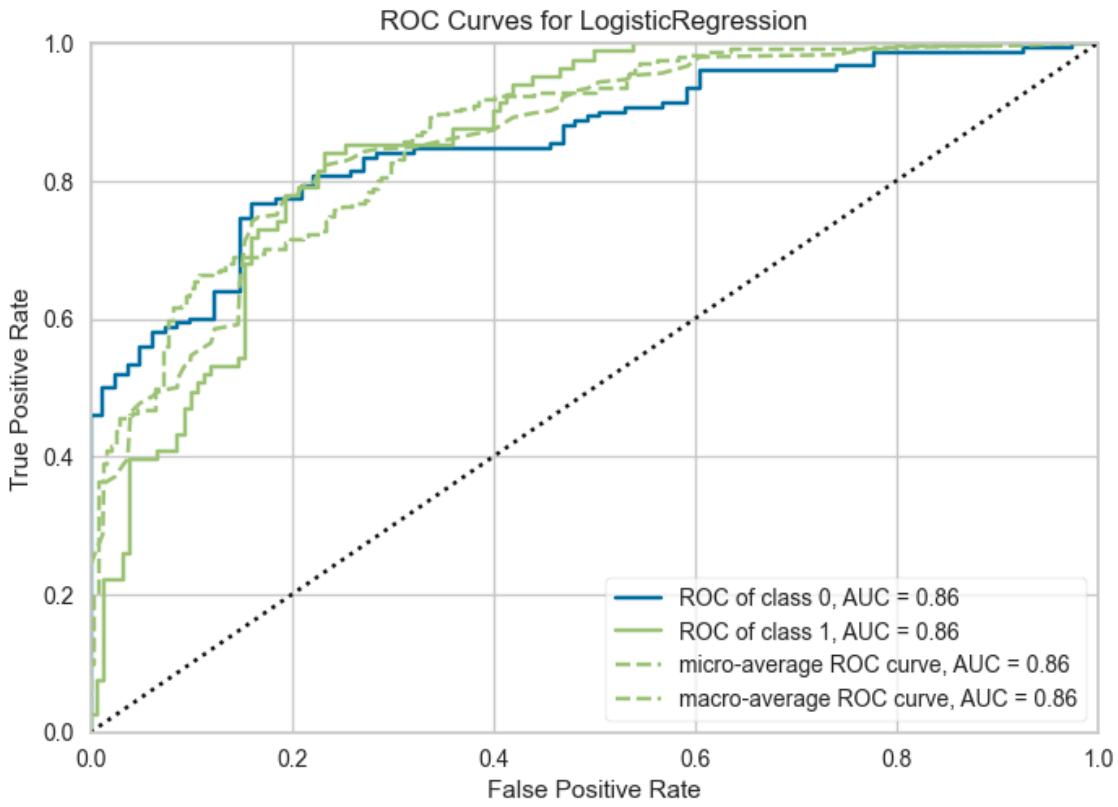
When you call `evaluate_model()`, it uses only the training data (the data you used in the setup() function) to assess the model's performance. This is crucial for avoiding any bias that may arise from using test data.

In [15]:
```
evaluate_model(best)
```

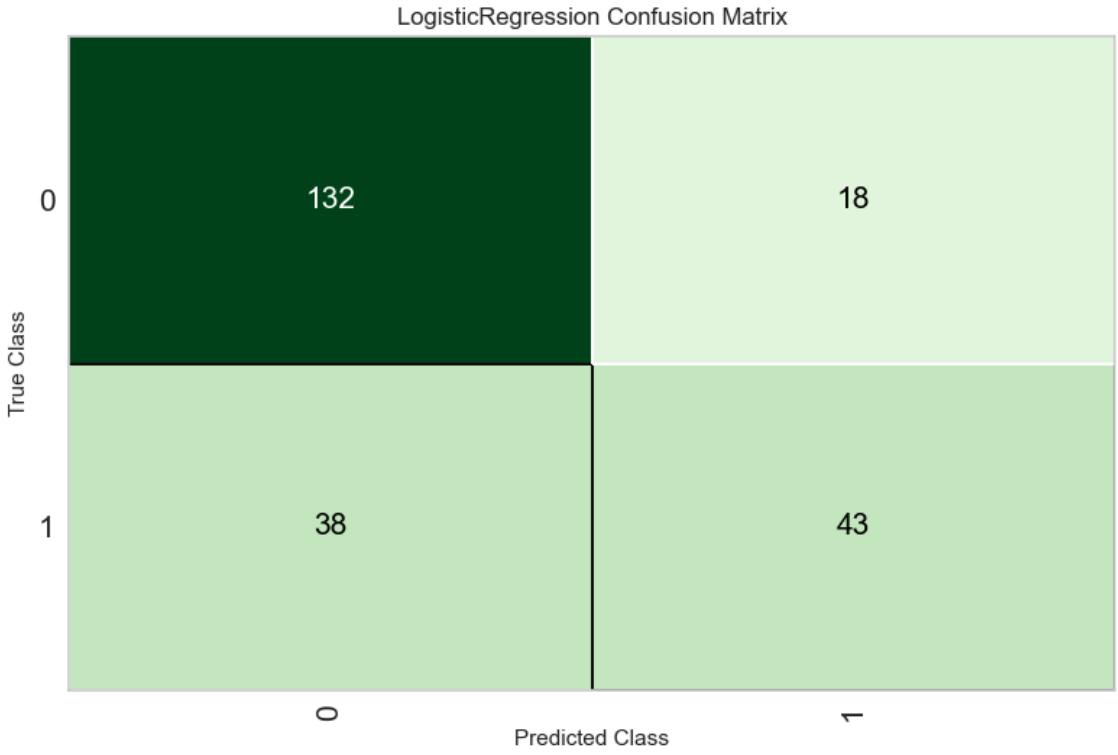
```
interactive(children=(ToggleButtons(description='Plot Type:', ico
ns=('',), options=(('Pipeline Plot', 'pipelin…
```

`evaluate_model()` can only be used in Notebook since it uses `ipywidget`. You can also use the plot_model function to generate plots individually.

In [21]:
```
plot_model(best, plot = 'auc')
```



In [22]:
```
plot_model(best, plot = 'confusion_matrix')
```



finalize_model()

The finalize_model() function in PyCaret is used to finalize a trained model. This means that after you have trained and evaluated a model using cross-validation (with create_model() or compare_models()), you can use finalize_model() to train the model on the entire dataset (including the test split).

In other words, the model is refit on the entire training dataset, so it is ready for deployment or making predictions on new data.

It retrains the model on the entire dataset, including both the training and test splits (used during cross-validation).

Once you finalize the model, it is in its final state and can be used for predictions on unseen data (like a test set or future predictions).

After finalizing, the model is no longer tuned or cross-validated. It's ready for real-world use.

Cross-validation models use only a portion of your dataset during training. When you use `finalize_model()`, it ensures the model is trained on all available data, giving it the maximum exposure to the patterns in the dataset.

Before making predictions: This is especially important if you plan to make predictions on new data or deploy the model in production.

Before finalizing, you typically evaluate your model using `evaluate_model()`, which performs cross-validation on the training data. This way, you can assess how well the model generalizes to unseen data without biasing it with the test set.

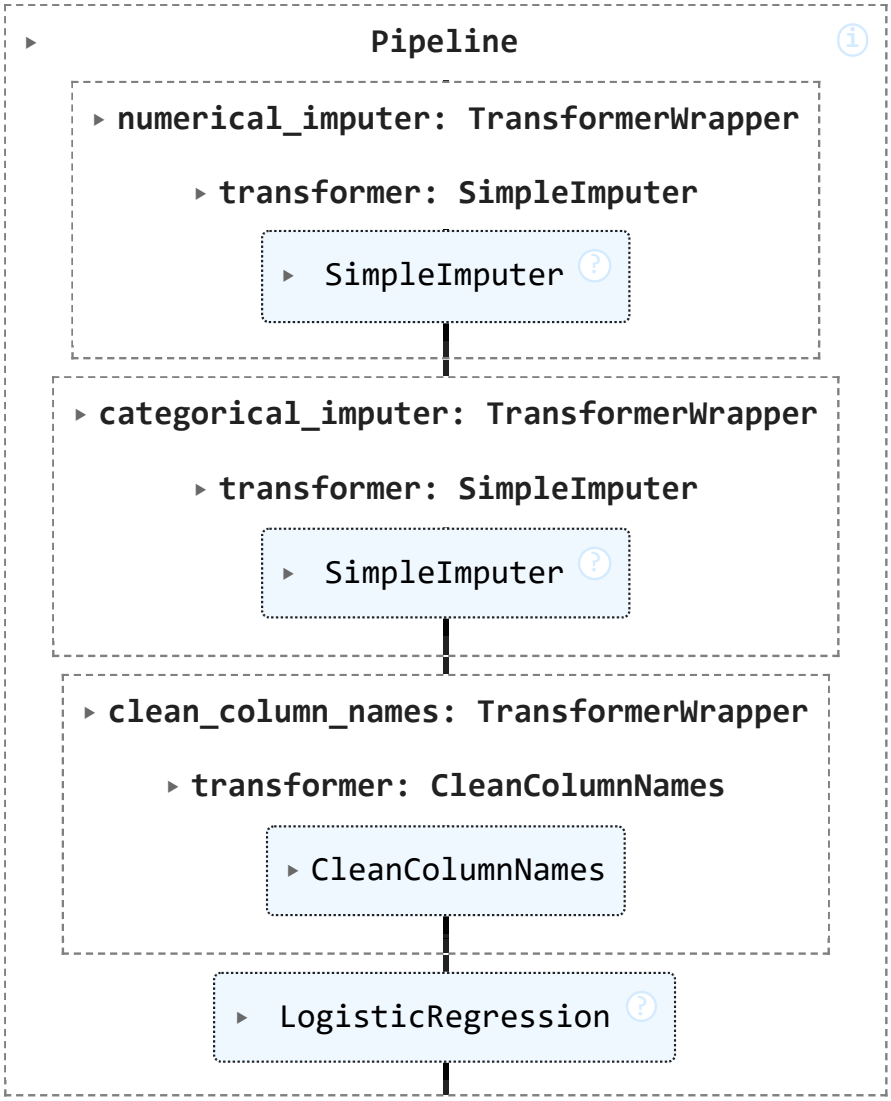
Once the model is finalized (after training on all data), you should not evaluate it against the test set again in the same way. Instead, you can use the test set to make predictions and then assess those predictions independently to gauge the model's performance on new, unseen data.

When to use `finalize_model()`:

- After Model Selection: Once you've chosen the best model using `compare_models()` or `create_model()`, and you're satisfied with the performance, you should call `finalize_model()` to fit the model on the full dataset.
- For Deployment: It's especially useful before you save the model for deployment (e.g., exporting with `save_model()`).

In [19]:
```
# Step 4: Finalize the model
finalize_model(best)  # Retrains on the full dataset
```

Out[19]:



predict_model()

The `predict_model()` function in PyCaret is used to generate predictions for new or unseen data using a trained model. This function allows you to evaluate how well your model performs on data it hasn't seen during training or cross-validation.

Before making predictions, `predict_model()` automatically applies any necessary preprocessing steps (like scaling, encoding, etc.) that were defined in the setup() phase.

This function scores the data and returns `prediction_label` and `prediction_score` probability of the predicted class). When data is None, it predicts label and score on the test set (created during the setup function).

In [20]:
```
predict_model(best)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	Logistic Regression	0.7576	0.8568	0.5309	0.7049	0.6056	0.4356	0.4447

Out[20]:

	Number of times pregnant	Plasma glucose concentration a 2 hours in an oral glucose tolerance test	Diastolic blood pressure (mm Hg)	Triceps skin fold thickness (mm)	2-Hour serum insulin (mu U/ml)	Body mass index (weight in kg/(height in m)^2)
552	6	114	88	0	0	27.799999
438	1	97	70	15	0	18.200001
149	2	90	70	17	0	27.299999
373	2	105	58	40	94	34.900002
36	11	138	76	0	0	33.200001
...	...	...	...	...	...	..
85	2	110	74	29	125	32.400002
7	10	115	0	0	0	35.299999
298	14	100	78	25	184	36.599998
341	1	95	74	21	73	25.900000
472	0	119	66	27	0	38.799999

231 rows × 11 columns

The evaluation metrics are calculated on the test set. The second output is the `pd.DataFrame` with predictions on the test set (see the last two columns). To generate labels on the unseen (new) dataset, simply pass the dataset in the data parameter under `predict_model` function.

Suppose i want to do prediction on the first 10 rows of `data`

NOTE : We can also load a diabetes data with same features using pandas dataframe using `read_csv()` and do prediction on it

In [29]:
```
test = data.head(10)

predict_model(best, data=test)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	Logistic Regression	0.7000	0.7500	0.6667	0.8000	0.7273	0.4000	0.4082

Out[29]:

	Number of times pregnant	Plasma glucose concentration a 2 hours in an oral glucose tolerance test	Diastolic blood pressure (mm Hg)	Triceps skin fold thickness (mm)	2-Hour serum insulin (mu U/ml)	Body mass index (weight in kg/(height in m)^2)
0	6	148	72	35	0	33.599998
1	1	85	66	29	0	26.600000
2	8	183	64	0	0	23.299999
3	1	89	66	23	94	28.100000
4	0	137	40	35	168	43.099998
5	5	116	74	0	0	25.600000
6	3	78	50	32	88	31.000000
7	10	115	0	0	0	35.299999
8	2	197	70	45	543	30.500000
9	8	125	96	0	0	0.000000

NOTE : Score means the probability of the predicted class (NOT the positive class). If prediction_label is 0 and prediction_score is 0.90, this means 90% probability of class 0. If you want to see the probability of both the classes, simply pass `raw_score=True` in the `predict_model()` function.

In [30]:
```
predict_model(best, data=test, raw_score=True)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	Logistic Regression	0.7000	0.7500	0.6667	0.8000	0.7273	0.4000	0.4082

Out[30]:

	Number of times pregnant	Plasma glucose concentration a 2 hours in an oral glucose tolerance test	Diastolic blood pressure (mm Hg)	Triceps skin fold thickness (mm)	2-Hour serum insulin (mu U/ml)	Body mass index (weight in kg/(height in m)^2)
0	6	148	72	35	0	33.599998
1	1	85	66	29	0	26.600000
2	8	183	64	0	0	23.299999
3	1	89	66	23	94	28.100000
4	0	137	40	35	168	43.099998
5	5	116	74	0	0	25.600000
6	3	78	50	32	88	31.000000
7	10	115	0	0	0	35.299999
8	2	197	70	45	543	30.500000
9	8	125	96	0	0	0.000000

Save the model

In [31]:
```
save_model(best, model_name='my_best_model')
```

```
Transformation Pipeline and Model Successfully Saved
```

Out[31]:
```
(Pipeline(memory=Memory(location=None),
          steps=[('numerical_imputer',
                  TransformerWrapper(exclude=None,
                                     include=['Number of times
pregnant',
                                              'Plasma glucose c
oncentration a 2 '
                                              'hours in an oral
glucose '
                                              'tolerance test',
                                              'Diastolic blood
pressure (mm Hg)',
                                              'Triceps skin fol
d thickness (mm)',
                                              '2-Hour serum ins
ulin (mu U/ml)',
                                              'Body mass index
(weight in '
                                              'kg/(height in m)
^2)',
                                              'Diabetes pedigr
e...
                  TransformerWrapper(exclude=None, include=Non
e,
                                     transformer=CleanColumnNam
es(match='[\\]\\[\\,\\{\\}\\"\\:]+'))),
                 ('trained_model',
                  LogisticRegression(C=1.0, class_weight=None,
dual=False,
                                     fit_intercept=True, interc
ept_scaling=1,
                                     l1_ratio=None, max_iter=10
00,
                                     multi_class='auto', n_jobs
=None,
                                     penalty='l2', random_state
=123,
                                     solver='lbfgs', tol=0.000
1, verbose=0,
                                     warm_start=False))],
          verbose=False),
 'my_best_model.pkl')
```

This saves the model in the current working directory as pickle file

Load the model back in environment

In [32]:
```
loaded_model = load_model('my_best_model')
```

```
Transformation Pipeline and Model Successfully Loaded
```

Now will try doing predictions on first 5 rows

In [35]:
```
loaded_model.predict(data.drop(['Class variable'], axis=1).head(
```

Out[35]:
```
array([1, 0, 1, 0, 1], dtype=int8)
```

Now this was for Classification but This whole same process can be done for Regression problem tasks as well