Basic ML Guide

This notebook covers developing and tuning simple `Classification` and `Regression` machine learning models in python.

- Dataset used for analysis is Sarah Gets a Diamond taken from University of Virginia, Darden Business Publishing.
- The regression model predicts the price(Y variable) of a diamond based on its multiple physical attributes/features(X variables). For the classification model, I have changed the definition of the Y variable. For all diamonds with price greater than $10,000 High/H label was added and for the remaining Low/L label was added.
- Basic data cleaning, data transformations, and pre-processing techniques have been applied on the on model development data. Model developement and Hyper-parameter tuning has been carried out for both the Regression and Classification models.

In [1]:

```
import pandas as pd
import numpy as np
import plotly.express as px
import seaborn as sns
import matplotlib.pyplot as plt
import shap

import xgboost as xgb
import lightgbm as lgb

from scipy.stats import gaussian_kde
from sklearn.svm import SVR, SVC
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder, MinMaxScaler
from sklearn.model_selection import KFold, cross_validate, train_test_split, cross_val_score,RandomizedSearchCV
from sklearn.utils import check_array

from sklearn.metrics import make_scorer, r2_score, mean_absolute_error, roc_curve, auc, classification_report
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, precision_score, recall_score, accuracy_score

import warnings

warnings.filterwarnings('ignore')
pd.set_option('display.float_format', lambda x : '%.5f' % x)
```

```
pandas.Int64Index is deprecated and will be removed from pandas in a future version. Use pandas.Index with the appropriat
e dtype instead.
```

1. Defining utility functions to be used later

In [2]:

```
# function to prepare Confusion Matrix, RoC-AUC curve, and other relvant statistics for a Classification problem

def clf_report(clf, x_true, y_true, split):
    y_pred = clf.predict(x_true)
    probs = clf.predict_proba(x_true)
    print('Classification report for {} data'.format(split))
    cm = confusion_matrix(y_true, y_pred, labels=clf.classes_)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
    disp.plot()
    plt.show()
    print('Overall Accuracy : {}'.format(round(accuracy_score(y_true, y_pred) * 100, 2)))
    print('Precision Score : {}'.format(round(precision_score(y_true, y_pred, average='binary') * 100, 2)))
    print('Recall Score : {}'.format(round(recall_score(y_true, y_pred, average='binary') * 100, 2)))
    preds = probs[:,1]
    fpr, tpr, threshold = roc_curve(y_true, preds)
    roc_auc = auc(fpr, tpr)
    print('AUC : {}'.format(round(roc_auc * 100, 2)))
    plt.figure()
    plt.plot(fpr, tpr, label='AUC = %0.2f)' % roc_auc)
    plt.plot([0.0, 1.0], [0, 1],'r--')
    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.1, 1.1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('RoC-AUC on {} Data'.format(split))
    plt.legend(loc="lower right")
    plt.show()
    print('\n')
```

In [3]:

```
# functions to calculate MAPE and Negative-MAPE for Regression problems

def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

def neg_mean_absolute_percentage_error(y_true, y_pred):
    return (-1)*(mean_absolute_percentage_error(y_true, y_pred))
```

In [4]:

```
# function to check a Regression model performanace on Train data
```

```python
# Performance measured on R2, MAE, MAPE metrics on 10-fold Cross-Validation

def train_metrics(rgs, X_cv, Y_cv):
    print('Cross Validated Metric Results for Train Data:')
    kf = KFold(n_splits=10, random_state=25, shuffle=True)
    metric_df = pd.DataFrame()

    for train_index, test_index in kf.split(X_cv):
        x_train, x_test = X_cv.iloc[train_index], X_cv.iloc[test_index]
        y_train, y_test = Y_cv.iloc[train_index], Y_cv.iloc[test_index]
        y_pred = rgs.fit(x_train, y_train).predict(x_test)
        r2 = r2_score(y_test, y_pred)
        mae = mean_absolute_error(y_test, y_pred)
        mape = mean_absolute_percentage_error(y_test, y_pred)
        temp_df = pd.DataFrame([[r2, mae, mape]])
        metric_df = pd.concat([metric_df, temp_df], ignore_index = True)

    metric_df.loc['Mean'] = round((metric_df.mean()),2)
    metric_df.loc['Std Dev'] = round((metric_df.std()),2)
    metric_df = metric_df.set_axis(['R-sq', 'MAE', 'MAPE'], axis=1, inplace=False)

    with pd.option_context('float_format', '{:.2f}'.format, 'display.expand_frame_repr', False):
        print(metric_df,'\n')
```

In [5]:

```python
# function to check a Regression model performanace on Test data
# Performance measured on R2, MAE, MAPE metrics on 10-fold Cross-Validation

def test_metrics(Y_test, Y_pred):
    print('Metric Results for Test Data:')
    r2 = r2_score(Y_test, Y_pred)
    mae = mean_absolute_error(Y_test, Y_pred)
    mape = mean_absolute_percentage_error(Y_test, Y_pred)
    metric_df = pd.DataFrame([[r2, mae, mape]], columns=['R-sq', 'MAE', 'MAPE'])
    with pd.option_context('float_format', '{:.2f}'.format, 'display.expand_frame_repr', False):
        print(metric_df,'\n')
```

2. Data import

In [6]:

```python
df_full = pd.read_csv('./sgd.csv')
display(df_full.shape)
```

```
display(df_full.head())
display(df_full.tail())
```

```
(9142, 9)
```

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	5169.00000
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	3470.00000
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	3183.00000
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	4370.00000
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	3171.00000

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
9137	9138	0.96000	Ideal	F	SI1	EX	EX	GIA	NaN
9138	9139	1.02000	Very Good	E	VVS1	EX	G	GIA	NaN
9139	9140	1.51000	Good	I	VS1	G	G	GIA	NaN
9140	9141	1.24000	Ideal	H	VS2	VG	VG	GIA	NaN
9141	9142	0.79000	Ideal	I	VS1	EX	EX	GIA	NaN

In [7]:

```
# selecting the rows with non-null values for the target/Y variable

df = df_full.copy()
df = df[:6000]
display(df.shape)
display(df.head())
display(df.tail())
```

```
(6000, 9)
```

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	5169.00000
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	3470.00000
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	3183.00000

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	4370.00000
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	3171.00000

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
5995	5996	1.03000	Ideal	D	SI1	EX	EX	GIA	6250.00000
5996	5997	1.00000	Very Good	D	SI1	VG	VG	GIA	5328.00000
5997	5998	1.02000	Ideal	D	SI1	EX	EX	GIA	6157.00000
5998	5999	1.27000	Signature-Ideal	G	VS1	EX	EX	GIA	11206.00000
5999	6000	2.19000	Ideal	E	VS1	EX	EX	GIA	30507.00000

In [8]:

```
# descriptive statistics for categorical variables

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

Out[8]:

	Cut	Color	Clarity	Polish	Symmetry	Report
count	6000	6000	6000	6000	6000	6000
unique	5	6	7	4	4	2
top	Ideal	G	SI1	EX	VG	GIA
freq	2482	1501	2059	2425	2417	5266

In [9]:

```
# descriptive statistics for numeric variables across 3 terciles

df.describe(include='number', percentiles=[0.33,0.66])
```

Out[9]:

	ID	Carat Weight	Price
count	6000.00000	6000.00000	6000.00000
mean	3000.50000	1.33452	11791.57933
std	1732.19514	0.47570	10184.35005

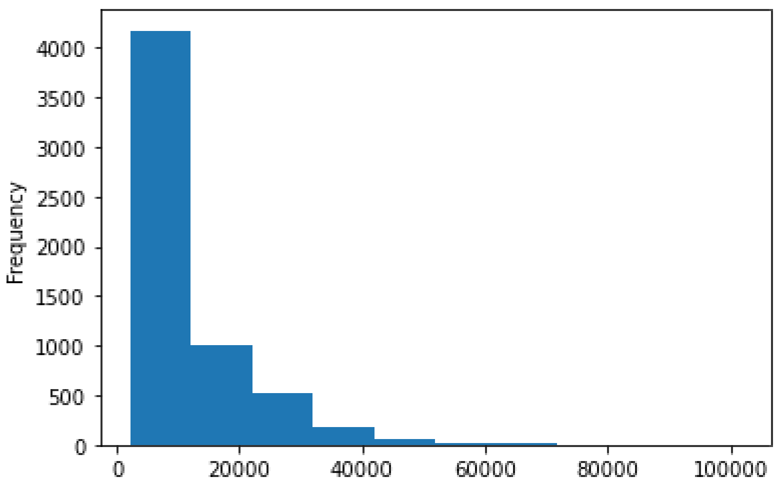
	ID	Carat Weight	Price
min	1.00000	0.75000	2184.00000
33%	1980.67000	1.02000	5789.00000
50%	3000.50000	1.13000	7857.00000
66%	3960.34000	1.50000	11083.36000
max	6000.00000	2.91000	101561.00000

In [10]:

```
# checking distribution for Price variable.
# This will help in creating a new target variable definition for the classification problem

df['Price'].plot.hist()
```

Out[10]:

```
<AxesSubplot:ylabel='Frequency'>
```



In [11]:

```
count = df[df['Price'] < 10000].count()
print(count)
```

```
ID              3657
Carat Weight    3657
Cut             3657
Color           3657
Clarity         3657
```

```
Polish          3657
Symmetry        3657
Report          3657
Price           3657
dtype: int64
```

In [12]:

```python
# creating a new variable/feature - Rate. This will serve as the target variable for the classification problem

df['Rate'] = np.where(((df['Price'] <= 10000)), 'L', 'H')
display(df.head())
```

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price	Rate
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	5169.00000	L
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	3470.00000	L
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	3183.00000	L
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	4370.00000	L
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	3171.00000	L

In [13]:

```python
df['Rate'].value_counts(), df['Rate'].value_counts().sum()
```

Out[13]:

```
(L    3658
 H    2342
 Name: Rate, dtype: int64,
 6000)
```

In [14]:

```python
# dataset for the classification problem

df_clf = df.copy()
df_clf = df_clf.drop('Price', axis=1)
df_clf.head()
```

Out[14]:

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Rate
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	L
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	L
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	L

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Rate
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	L
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	L

In [15]:

```
# dataset for the regression problem

df_reg = df.copy()
df_reg = df_reg.drop('Rate', axis=1)
df_reg.head()
```

Out[15]:

	ID	Carat Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	5169.00000
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	3470.00000
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	3183.00000
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	4370.00000
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	3171.00000

3. Classification

3.1 Data Cleaning and Exploratory Analysis

In [16]:

```
# descriptive statistics for categorical variables

df_clf.describe(include='object')
```

Out[16]:

	Cut	Color	Clarity	Polish	Symmetry	Report	Rate
count	6000	6000	6000	6000	6000	6000	6000
unique	5	6	7	4	4	2	2
top	Ideal	G	SI1	EX	VG	GIA	L
freq	2482	1501	2059	2425	2417	5266	3658

In [17]:

```
# descriptive statistics for numeric variables

df_clf.describe(include='number')
```

Out[17]:

	ID	Carat Weight
count	6000.00000	6000.00000
mean	3000.50000	1.33452
std	1732.19514	0.47570
min	1.00000	0.75000
25%	1500.75000	1.00000
50%	3000.50000	1.13000
75%	4500.25000	1.59000
max	6000.00000	2.91000

In [18]:

```
# removing white spaces from the column names

dictionary = {' ' : '_', '-' : ''}
df_clf.replace(dictionary, regex=True, inplace=True)
df_clf.columns = df_clf.columns.str.replace(' ', '_')
display(df_clf.head())
display(df_clf.tail())
```

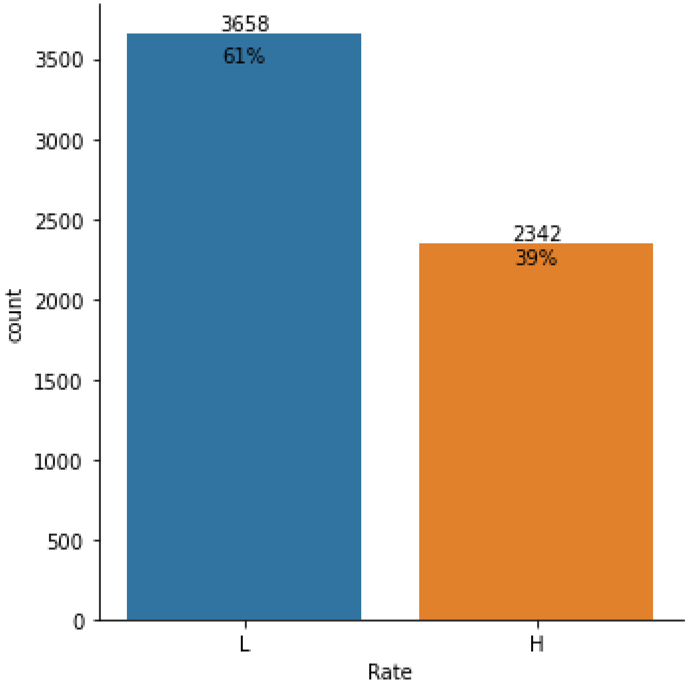
	ID	Carat_Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Rate
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	L
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	L
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	L
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	L
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	L

	ID	Carat_Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Rate

	ID	Carat_Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Rate
5995	5996	1.03000	Ideal	D	SI1	EX	EX	GIA	L
5996	5997	1.00000	Very_Good	D	SI1	VG	VG	GIA	L
5997	5998	1.02000	Ideal	D	SI1	EX	EX	GIA	L
5998	5999	1.27000	SignatureIdeal	G	VS1	EX	EX	GIA	H
5999	6000	2.19000	Ideal	E	VS1	EX	EX	GIA	H

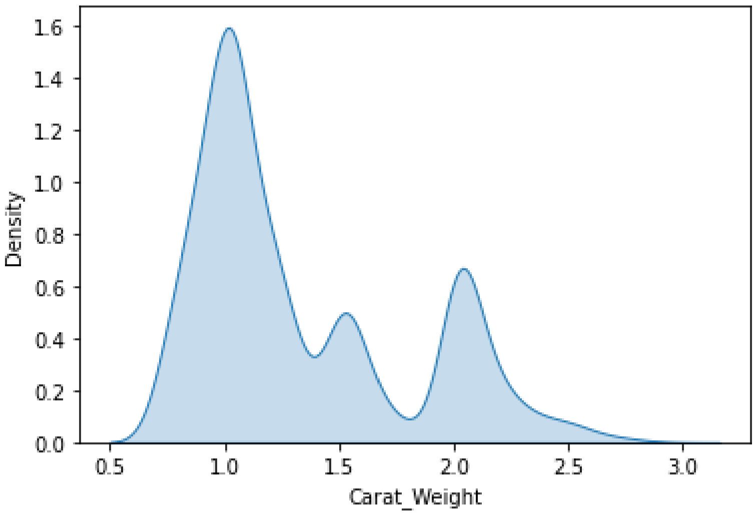
In [19]:

```
# Target Class/Variable distribution : Count and Percentage wise

grid = sns.catplot(x='Rate', kind='count', data=df_clf)
ax = grid.axes[0, 0]
ax.bar_label(ax.containers[0])
for p in ax.patches:
    percentage = '{}%'.format(round(100 * (p.get_height()) / (len(df_clf))),2)
    x = p.get_x() + (p.get_width())/2
    y = p.get_height() - 0.05 * (p.get_height())
    ax.annotate(percentage, (x, y), ha='center')
plt.show()
```



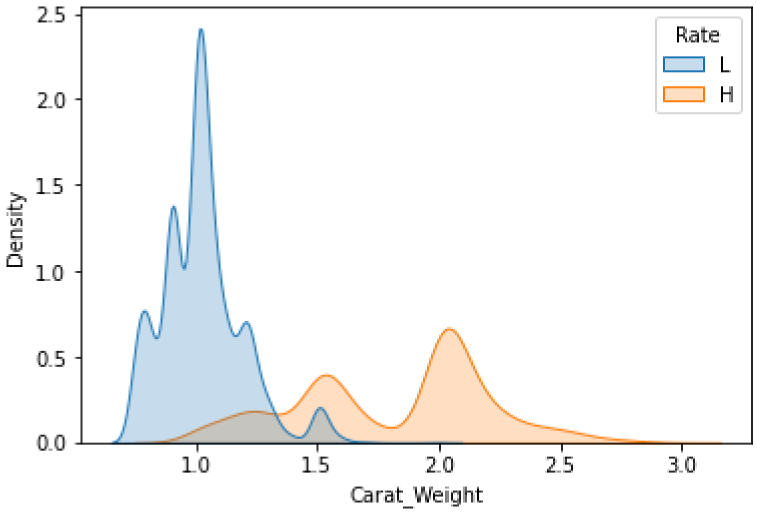
In [20]:

```
# density plot for Carat_Weight to see if some mathematical transformations are needed for this variable

sns.kdeplot(df_clf['Carat_Weight'], fill='bool')
plt.show()
```



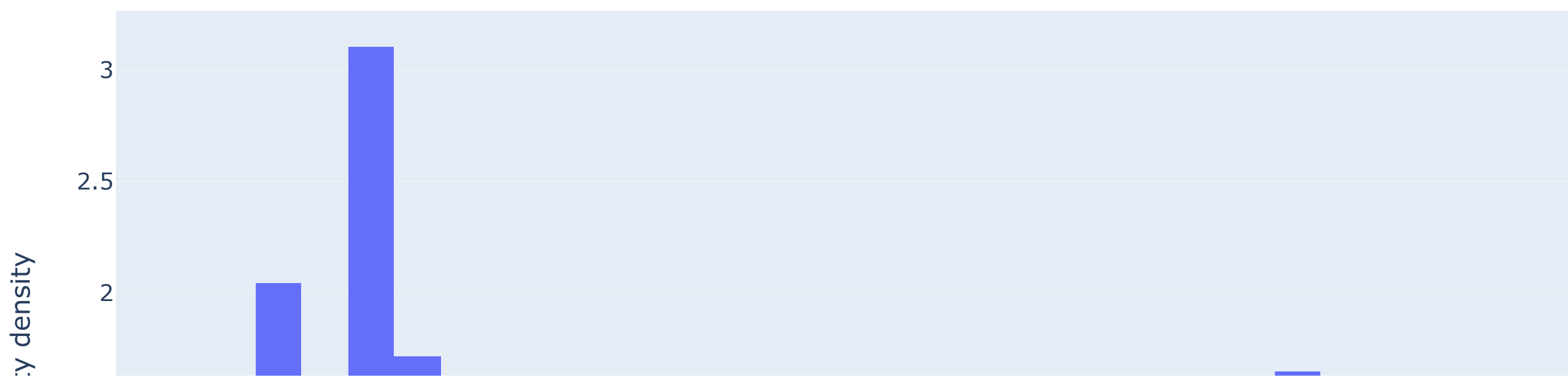
In [21]:

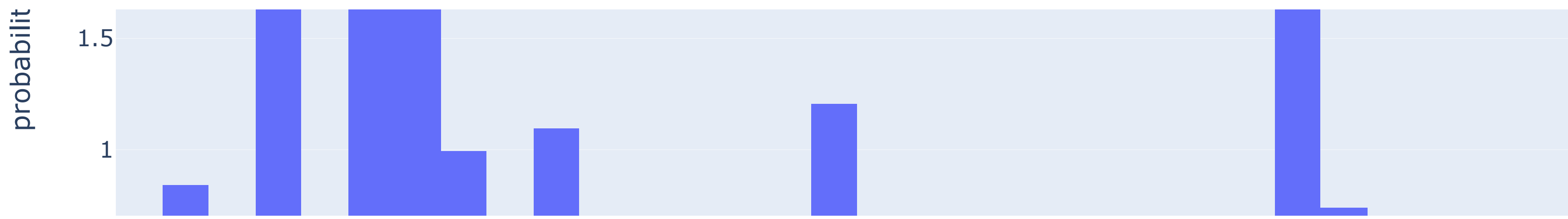
```python
# density plot for Carat_Weight across different Target classes. High Rate diamonds tend to have higher carat weight

sns.kdeplot(data=df_clf, x='Carat_Weight', hue='Rate', fill='bool')
plt.show()
```



In [22]:

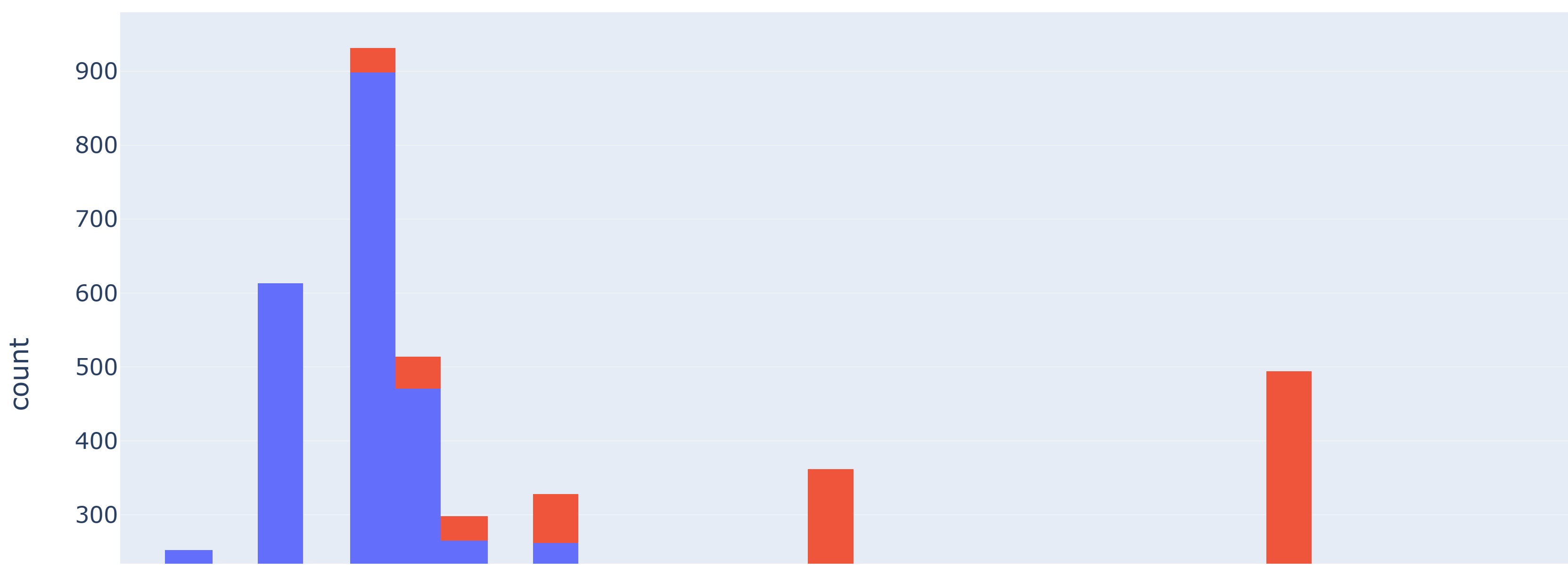
```python
# histogram plots with a different visualization library to check Carat_Weight distribution

fig = px.histogram(df_clf, x='Carat_Weight', histnorm='probability density')
fig.show()
```





In [23]:

```
# histogram plots with a different visualization library to check Carat_Weight distribution across different classes

fig = px.histogram(df_clf, x='Carat_Weight', color='Rate')
fig.show()
```



3.2 Pre-processing and Feature Engineering

In [24]:

```
# creating new features with some mathematical-transformation

df_clf['log_CW'] = np.log(df_clf['Carat_Weight'])
df_clf['norm_CW'] = MinMaxScaler().fit_transform(df_clf[['Carat_Weight']])
df_clf['lgnm_CW'] = np.log((df_clf['norm_CW']+0.00001))
```

In [25]:

```
df_clf.dtypes
```

Out[25]:

```
ID                int64
Carat_Weight    float64
Cut              object
Color            object
Clarity          object
Polish           object
Symmetry         object
Report           object
Rate             object
log_CW          float64
norm_CW         float64
lgnm_CW         float64
dtype: object
```

In [26]:

```
# collecting all the numerical and categorical variables and saving them in two different lists

categorical_columns_seen = [c for i,c in enumerate(df_clf.columns) if df_clf.dtypes[i] in [object]]
categorical_columns_seen.remove('Rate')
numerical_columns_seen = [c for i,c in enumerate(df_clf.columns) if df_clf.dtypes[i] not in [object]]
categorical_columns_seen, numerical_columns_seen
```

Out[26]:
```
(['Cut', 'Color', 'Clarity', 'Polish', 'Symmetry', 'Report'],
 ['ID', 'Carat_Weight', 'log_CW', 'norm_CW', 'lgnm_CW'])
```

In [27]:
```
# creating one hot encoded labels for categorical data

encoder = OneHotEncoder(handle_unknown='error', drop='first')
encoded_df = (pd.DataFrame(encoder.fit_transform(df_clf[categorical_columns_seen]).toarray())).astype(int)
encoded_df.columns = encoder.get_feature_names_out(categorical_columns_seen)
encoded_df.head()
```

Out[27]:

	Cut_Good	Cut_Ideal	Cut_SignatureIdeal	Cut_Very_Good	Color_E	Color_F	Color_G	Color_H	Color_I	Clarity_IF	...	Clarity_VS2	Clarity_VVS
0	0	1	0	0	0	0	0	1	0	0	...	0	
1	0	1	0	0	0	0	0	1	0	0	...	0	
2	0	1	0	0	0	0	0	1	0	0	...	0	
3	0	1	0	0	1	0	0	0	0	0	...	0	
4	0	1	0	0	0	0	1	0	0	0	...	0	

5 rows × 22 columns

In [28]:
```
tmp_df = pd.DataFrame()
tmp_df[numerical_columns_seen] = df_clf[numerical_columns_seen]
tmp_df.head()
```

Out[28]:

	ID	Carat_Weight	log_CW	norm_CW	lgnm_CW
0	1	1.10000	0.09531	0.16204	-1.81987
1	2	0.83000	-0.18633	0.03704	-3.29557
2	3	0.85000	-0.16252	0.04630	-3.07248
3	4	0.91000	-0.09431	0.07407	-2.60255
4	5	0.83000	-0.18633	0.03704	-3.29557

In [29]:
```
# converting Target Variable/Class into 1-0
```

```
cols = list(encoded_df.columns.values)
tmp_df[cols] = encoded_df[cols]
tmp_df['Rate'] = np.where(((df_clf['Rate'] == 'H')), 1, 0)
tmp_df.head()
```

Out[29]:

	ID	Carat_Weight	log_CW	norm_CW	lgnm_CW	Cut_Good	Cut_Ideal	Cut_SignatureIdeal	Cut_Very_Good	Color_E	...	Clarity_VVS1	Clarity
0	1	1.10000	0.09531	0.16204	-1.81987	0	1	0	0	0	...	0	
1	2	0.83000	-0.18633	0.03704	-3.29557	0	1	0	0	0	...	0	
2	3	0.85000	-0.16252	0.04630	-3.07248	0	1	0	0	0	...	0	
3	4	0.91000	-0.09431	0.07407	-2.60255	0	1	0	0	1	...	0	
4	5	0.83000	-0.18633	0.03704	-3.29557	0	1	0	0	0	...	0	

5 rows × 28 columns

3.3 Model Developement (feature set -1)

In [30]:

```
# creating Train and test data

Y_clf = tmp_df['Rate']
X_clf = tmp_df.drop(['Rate', 'ID', 'log_CW', 'norm_CW', 'lgnm_CW'], axis = 1)

X_train_clf, X_test_clf, Y_train_clf, Y_test_clf = train_test_split(X_clf, Y_clf, train_size = 0.8334, random_state = 25)
X_train_clf.shape, X_test_clf.shape
```

Out[30]:

```
((5000, 23), (1000, 23))
```

In [31]:

```
shap.initjs()
```

In [32]:

```
# model 1 : Light Gradient Boosting model

lgb_model = lgb.LGBMClassifier(boosting_type='goss', num_leaves=16, max_depth=-1, learning_rate=0.1, n_estimators=64, n_j
```

```
                               objective='binary', reg_lambda=0.5, importance_type='gain', silent=True, random_state=25)
lgb_model.fit(X_train_clf, Y_train_clf)
```

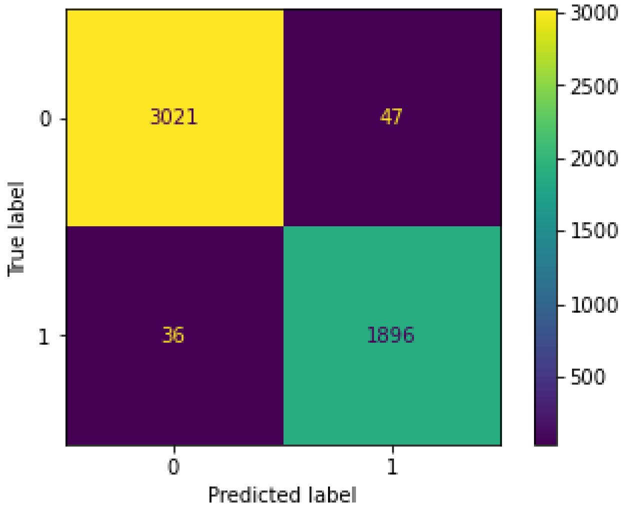
Out[32]:

```
LGBMClassifier(boosting_type='goss', importance_type='gain', n_estimators=64,
               num_leaves=16, objective='binary', random_state=25,
               reg_lambda=0.5)
```

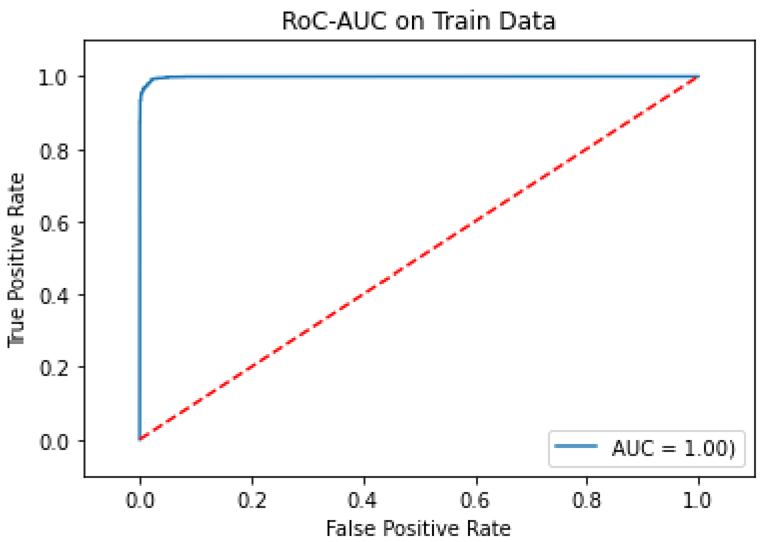
In [33]:

```
# checking model performance on train and test data

clf_report(lgb_model, X_train_clf, Y_train_clf, 'Train')
clf_report(lgb_model, X_test_clf, Y_test_clf, 'Test')
```

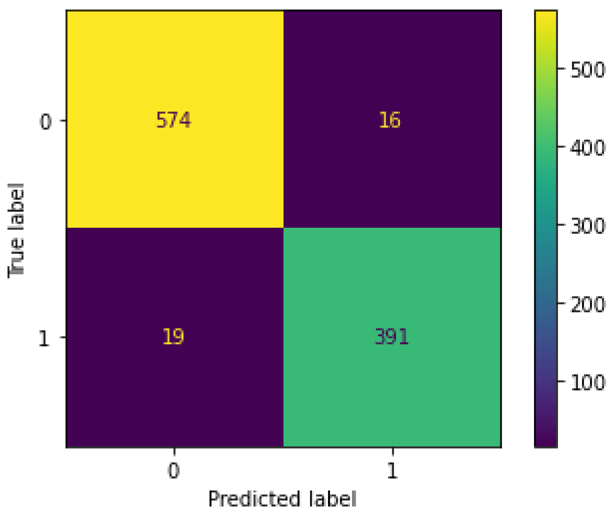
Classification report for Train data



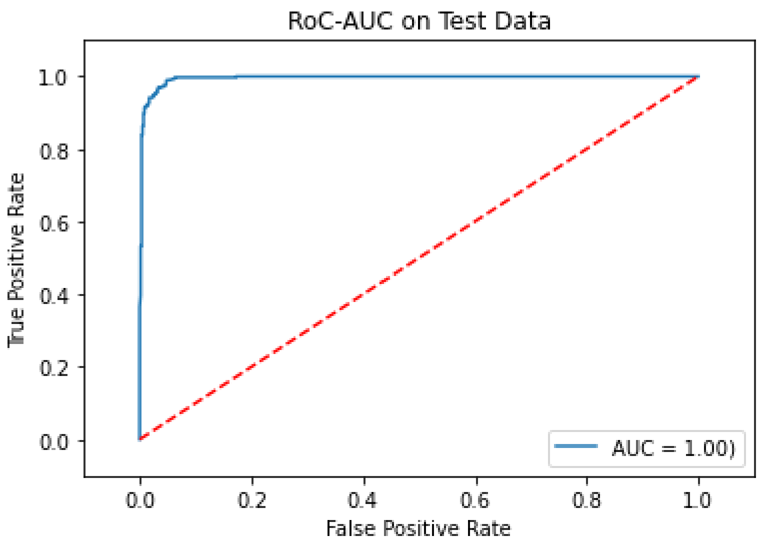

```
Overall Accuracy : 98.34
Precision Score : 97.58
Recall Score : 98.14
AUC : 99.91
```



Classification report for Test data

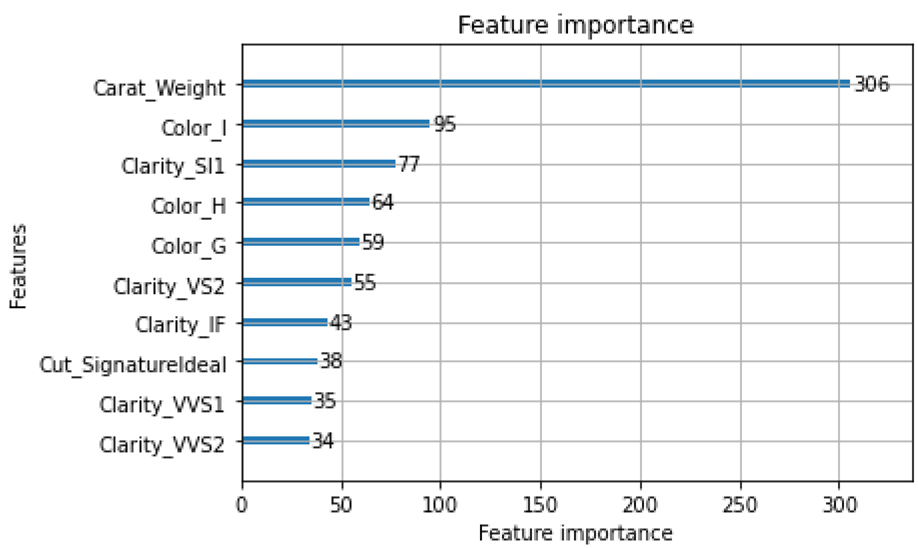



```
Overall Accuracy : 96.5
Precision Score : 96.07
Recall Score : 95.37
AUC : 99.52
```



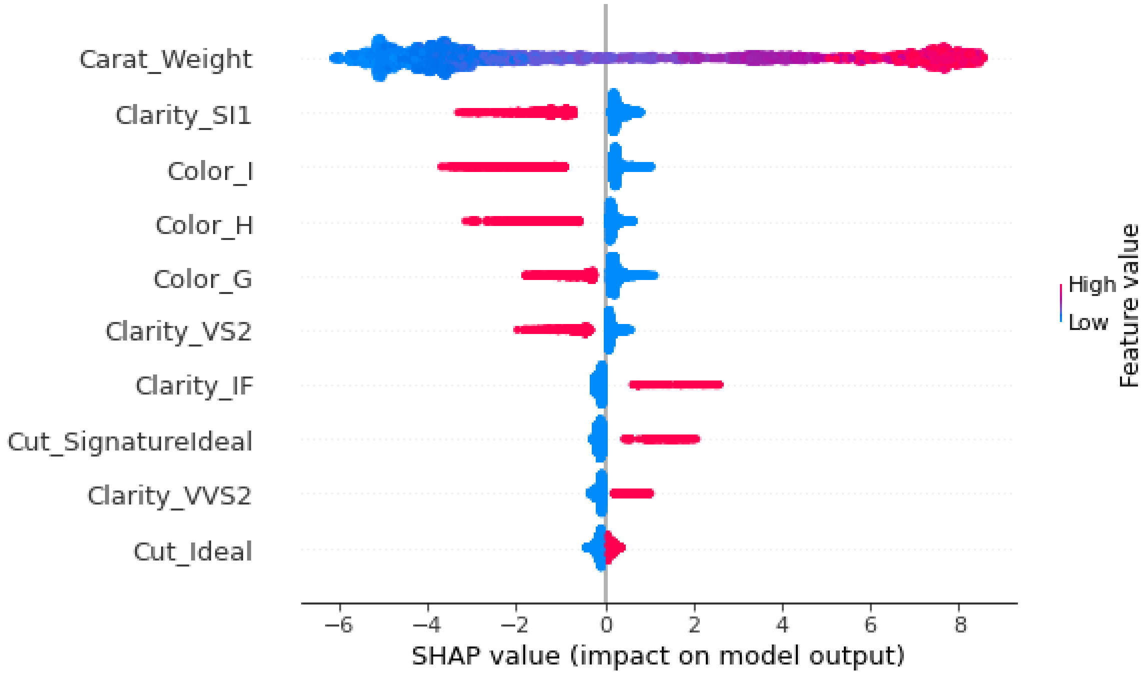
In [34]:
```
# creating feature importance plot

lgb.plot_importance(lgb_model, max_num_features=10)
```

Out[34]: <AxesSubplot:title={'center':'Feature importance'}, xlabel='Feature importance', ylabel='Features'>



In [35]:

```
# creating SHAP-explainer plot for all features

explainer = shap.TreeExplainer(lgb_model)
shap_values = explainer.shap_values(X_train_clf)
shap.summary_plot(shap_values[1], X_train_clf, max_display=10)
plt.show()
```



In [36]:

```
# creating SHAP-explainer plot for all features but for a single data row

shap.force_plot(explainer.expected_value[1], shap_values[1][0,:], X_train_clf.iloc[0, :])
```

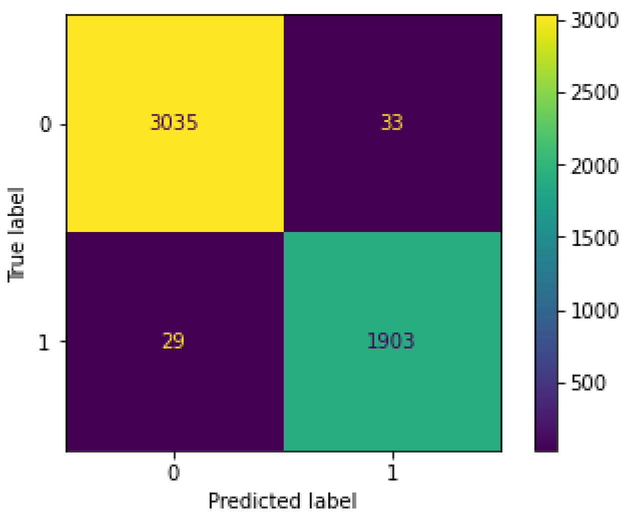
Out[36]:

In [37]:
```python
# model 2 : Xtreme Gradient Boosting model

xgb_model = xgb.XGBClassifier(n_estimators=64, max_depth=6, learning_rate=0.1, verbosity=0, use_label_encoder=False,
                              booster='gbtree', n_jobs=-1, reg_lambda=0.3, random_state=25)
xgb_model.fit(X_train_clf, Y_train_clf)
```

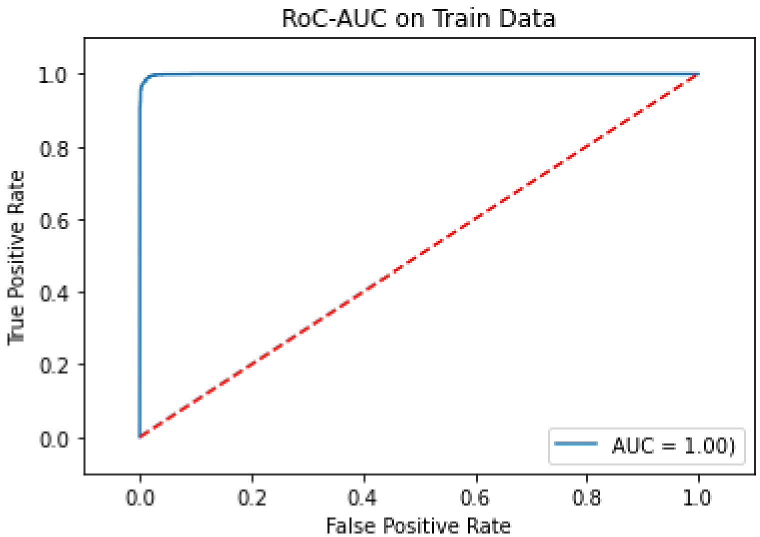
Out[37]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
              gamma=0, gpu_id=-1, importance_type=None,
              interaction_constraints='', learning_rate=0.1, max_delta_step=0,
              max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=64, n_jobs=-1,
              num_parallel_tree=1, predictor='auto', random_state=25,
              reg_alpha=0, reg_lambda=0.3, scale_pos_weight=1, subsample=1,
              tree_method='exact', use_label_encoder=False,
              validate_parameters=1, verbosity=0)
```

In [38]:
```python
# checking model performance on Train and test data

clf_report(xgb_model, X_train_clf, Y_train_clf, 'Train')
clf_report(xgb_model, X_test_clf, Y_test_clf, 'Test')
```

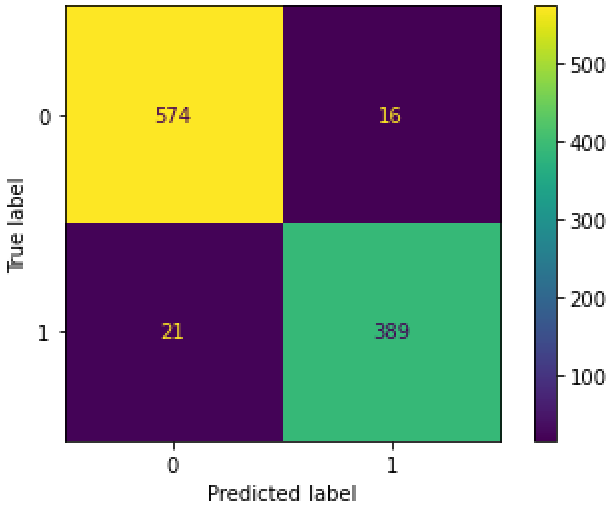
```
Classification report for Train data
```



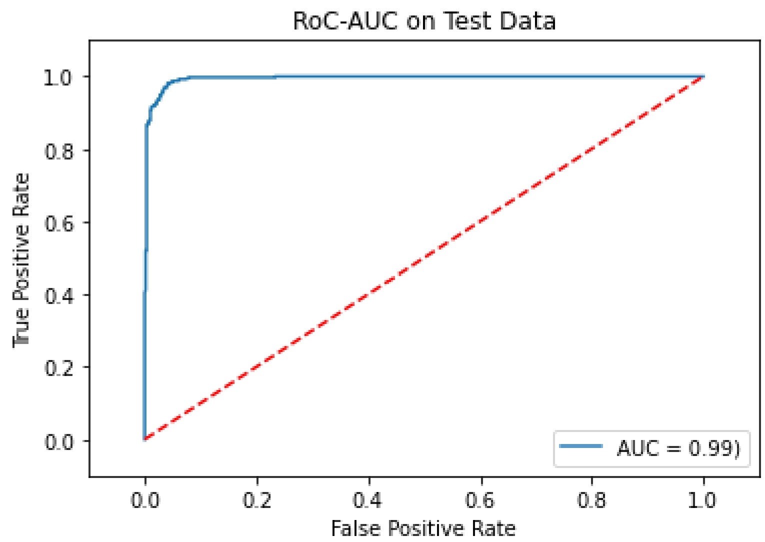

```
Overall Accuracy : 98.76
Precision Score : 98.3
Recall Score : 98.5
AUC : 99.94
```



Classification report for Test data

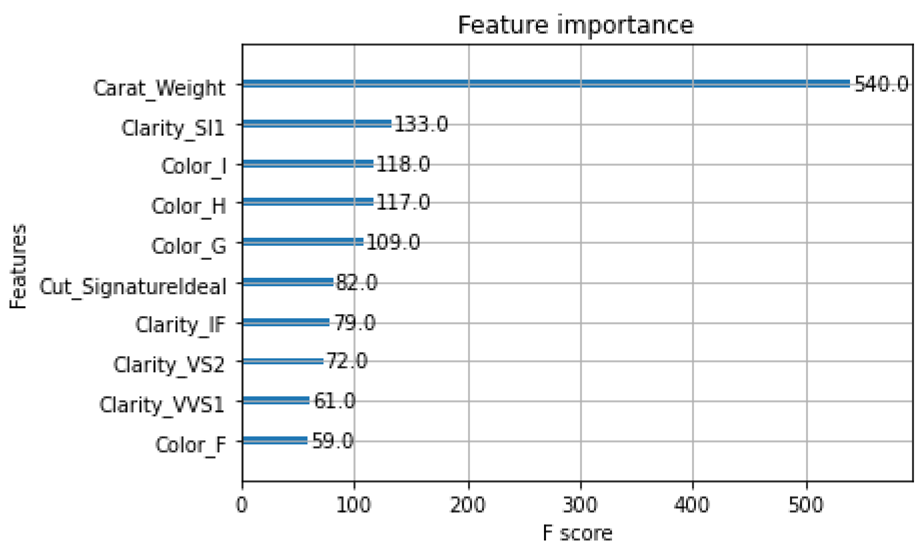


Overall Accuracy : 96.3
Precision Score : 96.05
Recall Score : 94.88
AUC : 99.47



In [39]:
```
# creating feature importance plot

xgb.plot_importance(xgb_model, max_num_features=10)
```

Out[39]:
```
<AxesSubplot:title={'center':'Feature importance'}, xlabel='F score', ylabel='Features'>
```

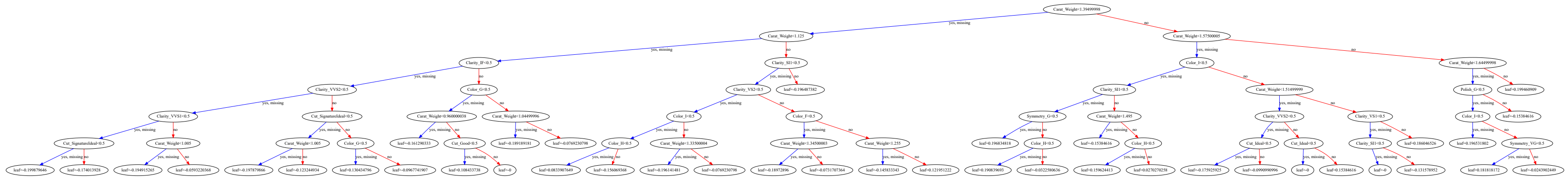


In [40]:

```python
# creating features tree plot to explain xgb-model decisions

xgb_params = xgb_model.get_booster()
for importance_type in ('weight', 'gain', 'cover', 'total_gain', 'total_cover'):
    print('%s: ' % importance_type, xgb_params.get_score(importance_type=importance_type))

xgb.to_graphviz(xgb_model, numtrees=0, size='20,20')
```

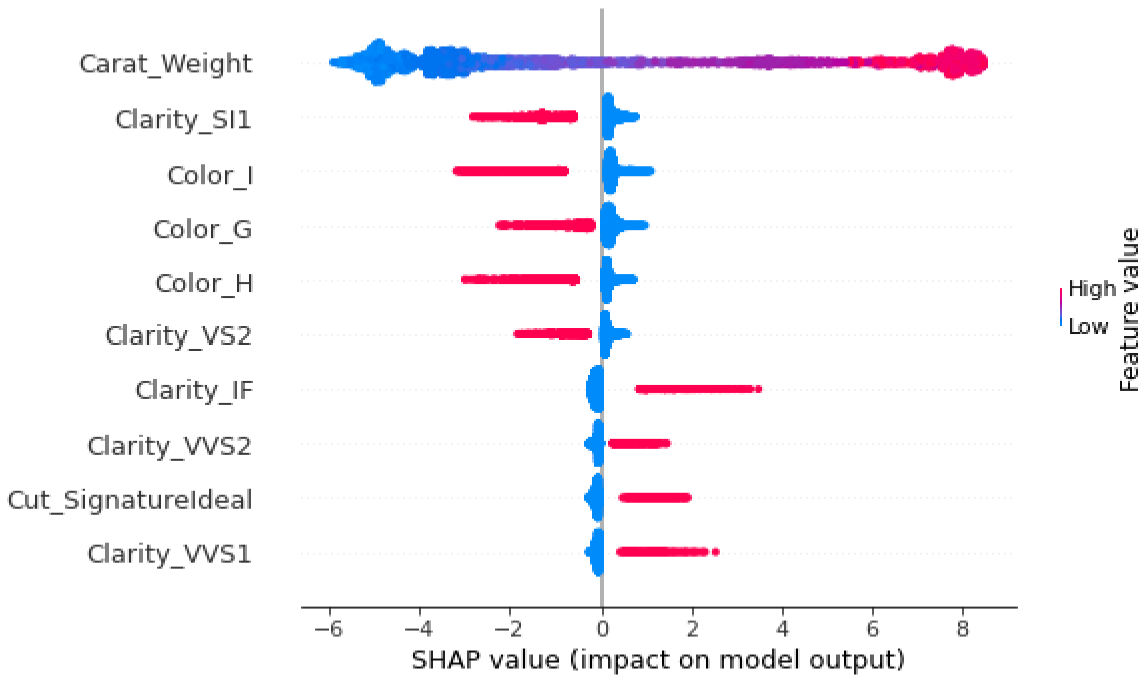
```
weight:  {'Carat_Weight': 540.0, 'Cut_Good': 16.0, 'Cut_Ideal': 52.0, 'Cut_SignatureIdeal': 82.0, 'Cut_Very_Good': 50.0, 'Color_E': 31.0, 'Color_F': 59.0, 'Color_G': 109.0, 'Color_H': 117.0, 'Color_I': 118.0, 'Clarity_IF': 79.0, 'Clarity_SI1': 133.0, 'Clarity_VS1': 44.0, 'Clarity_VS2': 72.0, 'Clarity_VVS1': 61.0, 'Clarity_VVS2': 57.0, 'Polish_G': 47.0, 'Polish_ID': 26.0, 'Polish_VG': 34.0, 'Symmetry_G': 27.0, 'Symmetry_ID': 8.0, 'Symmetry_VG': 48.0, 'Report_GIA': 10.0}
gain:  {'Carat_Weight': 46.63850402832031, 'Cut_Good': 1.3568055629730225, 'Cut_Ideal': 2.0487358570098877, 'Cut_SignatureIdeal': 3.1732118129730225, 'Cut_Very_Good': 1.293332576751709, 'Color_E': 4.7530059814453125, 'Color_F': 3.070007085800171, 'Color_G': 5.6871466636657715, 'Color_H': 8.126121520996094, 'Color_I': 13.49828815460205, 'Clarity_IF': 6.334937572479248, 'Clarity_SI1': 6.782624244689941, 'Clarity_VS1': 6.442461013793945, 'Clarity_VS2': 9.109304428100586, 'Clarity_VVS1': 4.946499824523926, 'Clarity_VVS2': 6.01350736618042, 'Polish_G': 2.2005317211151123, 'Polish_ID': 1.0057790279388428, 'Polish_VG': 1.5062283277511597, 'Symmetry_G': 1.3638572692871094, 'Symmetry_ID': 3.322267532348633, 'Symmetry_VG': 1.0099114179611206, 'Report_GIA': 1.8627281188964844}
cover:  {'Carat_Weight': 113.99594116210938, 'Cut_Good': 18.982723236083984, 'Cut_Ideal': 23.309589385986328, 'Cut_SignatureIdeal': 102.09420013427734, 'Cut_Very_Good': 10.433252334594727, 'Color_E': 24.423532485961914, 'Color_F': 21.392839431762695, 'Color_G': 34.65987014770508, 'Color_H': 41.618221282958984, 'Color_I': 58.61147689819336, 'Clarity_IF': 137.1781768798828, 'Clarity_SI1': 50.38148880004883, 'Clarity_VS1': 36.72146224975586, 'Clarity_VS2': 52.17066955566406, 'Clarity_VVS1': 148.55996704101562, 'Clarity_VVS2': 164.23777770996094, 'Polish_G': 28.90943717956543, 'Polish_ID': 8.962898254394531, 'Polish_VG': 17.54803466796875, 'Symmetry_G': 23.14409065246582, 'Symmetry_ID': 19.828092575073242, 'Symmetry_VG': 7.35402250289917, 'Report_GIA': 6.514835357666016}
total_gain:  {'Carat_Weight': 25184.79296875, 'Cut_Good': 21.70888900756836, 'Cut_Ideal': 106.53426361083984, 'Cut_SignatureIdeal': 260.203369140625, 'Cut_Very_Good': 64.6666259765625, 'Color_E': 147.3431854248047, 'Color_F': 181.1304168701172, 'Color_G': 619.8989868164062, 'Color_H': 950.7562255859375, 'Color_I': 1592.7979736328125, 'Clarity_IF': 500.4600830078125, 'Clarity_SI1': 902.0890502929688, 'Clarity_VS1': 283.4682922363281, 'Clarity_VS2': 655.8699340820312, 'Clarity_VVS1': 301.7364807128906, 'Clarity_VVS2': 342.7699279785156, 'Polish_G': 103.42498779296875, 'Polish_ID': 26.15025520324707, 'Polish_VG': 51.211761474609375, 'Symmetry_G': 36.82414627075195, 'Symmetry_ID': 26.578140258789062, 'Symmetry_VG': 48.474574996948242, 'Report_GIA': 18.627281188964844}
total_cover:  {'Carat_Weight': 61557.80859375, 'Cut_Good': 303.72357177734375, 'Cut_Ideal': 1212.0986328125, 'Cut_SignatureIdeal': 8371.724609375, 'Cut_Very_Good': 521.66259765625, 'Color_E': 757.1295166015625, 'Color_F': 1262.177490234375, 'Color_G': 3777.92578125, 'Color_H': 4869.33203125, 'Color_I': 6916.154296875, 'Clarity_IF': 10837.076171875, 'Clarity_SI1': 6700.73779296875, 'Clarity_VS1': 1615.744384765625, 'Clarity_VS2': 3756.2880859375, 'Clarity_VVS1': 9062.158203125, 'Clarity_VVS2': 9361.5537109375, 'Polish_G': 1358.7435302734375, 'Polish_ID': 233.0353546142578, 'Polish_VG': 596.6331787109375, 'Symmetry_G': 624.8904418945312, 'Symmetry_ID': 158.62474060058594, 'Symmetry_VG': 352.9930725097656, 'Report_GIA': 65.14835357666016}
```

Out[40]:



In [41]:

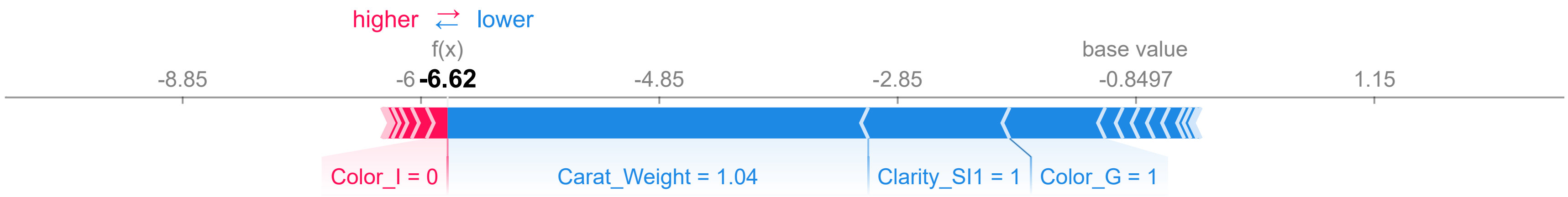
```python
# creating SHAP-explainer plot for all features

explainer = shap.TreeExplainer(xgb_model)
shap_values = explainer.shap_values(X_train_clf)
shap.summary_plot(shap_values, X_train_clf, max_display=10)
```



In [42]:

```python
# creating SHAP-explainer plot for all features but for a single data row

shap.force_plot(explainer.expected_value, shap_values[0,:], X_train_clf.iloc[0, :])
```

Out[42]:



3.4 Model Developement (feature set-2)

In [43]:
```
Y_clf = tmp_df['Rate']
X_clf = tmp_df.drop(['Rate', 'ID', 'lgnm_CW', 'log_CW', 'Carat_Weight'], axis = 1)

X_train_clf, X_test_clf, Y_train_clf, Y_test_clf = train_test_split(X_clf, Y_clf, train_size = 0.8334, random_state = 25)
X_train_clf.shape, X_test_clf.shape
```

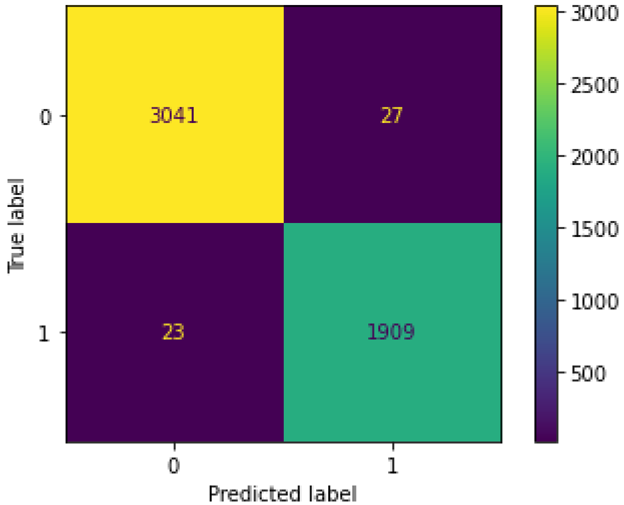
Out[43]:
```
((5000, 23), (1000, 23))
```

In [44]:
```
lgb_model = lgb.LGBMClassifier(boosting_type='goss', num_leaves=16, max_depth=-1, learning_rate=0.1, n_estimators=128, n_
                               objective='binary', reg_lambda=0.5, importance_type='gain', random_state=25, silent=True)
lgb_model.fit(X_train_clf, Y_train_clf)
```

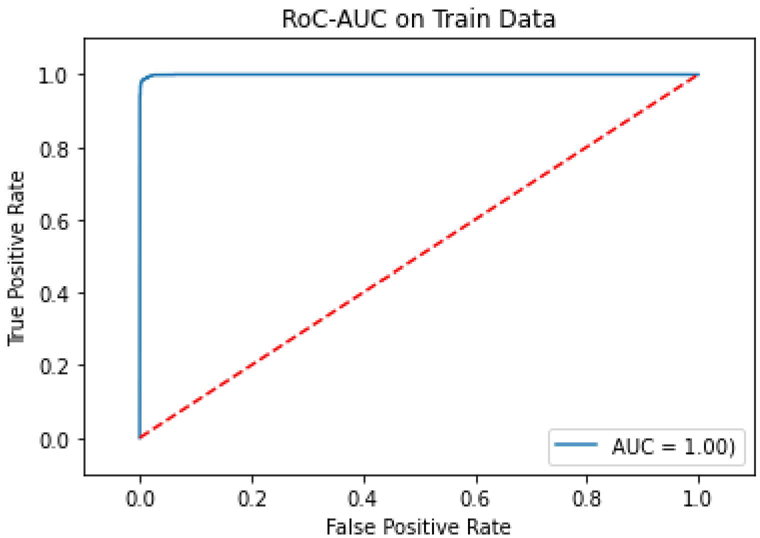
Out[44]:
```
LGBMClassifier(boosting_type='goss', importance_type='gain', n_estimators=128,
               num_leaves=16, objective='binary', random_state=25,
               reg_lambda=0.5)
```

In [45]:
```
clf_report(lgb_model, X_train_clf, Y_train_clf, 'Train')
clf_report(lgb_model, X_test_clf, Y_test_clf, 'Test')
```

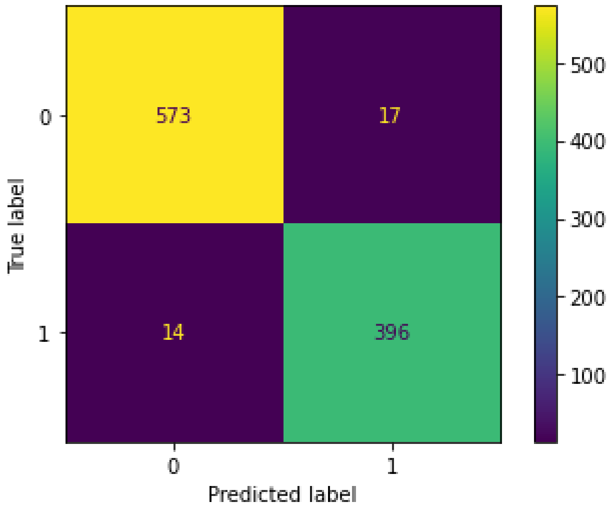
```
Classification report for Train data
```



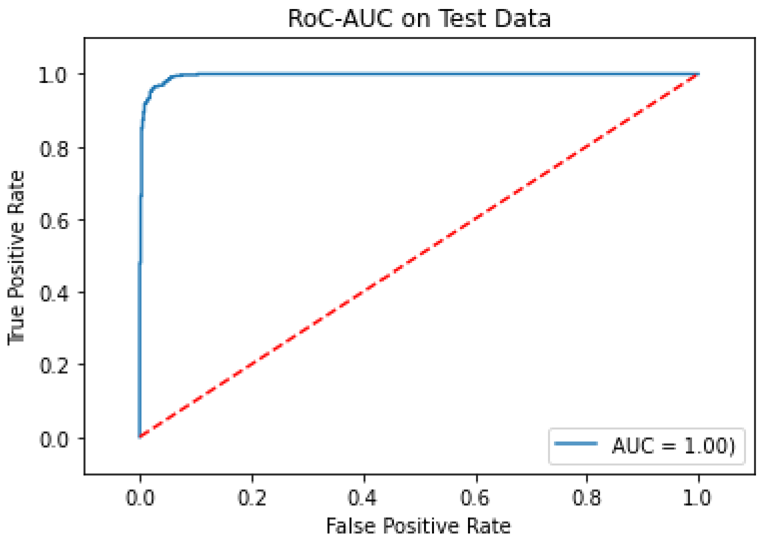

```
Overall Accuracy : 99.0
Precision Score : 98.61
Recall Score : 98.81
AUC : 99.97
```




```
Classification report for Test data
```




```
Overall Accuracy : 96.9
Precision Score : 95.88
Recall Score : 96.59
AUC : 99.58
```



3.5 Hyperparameter Tuning

In [46]:
```
params = {'n_estimators' : [64, 128, 256], 'max_depth' : [3,5,7], 'learning_rate' : [0.5, 0.1]}
randomized_object = RandomizedSearchCV(estimator=lgb.LGBMClassifier(), scoring='roc_auc', random_state=25,
```

```
                                        cv=10, n_jobs=-1, param_distributions=params, verbose=0)
randomized_object.fit(X_train_clf, Y_train_clf)
```

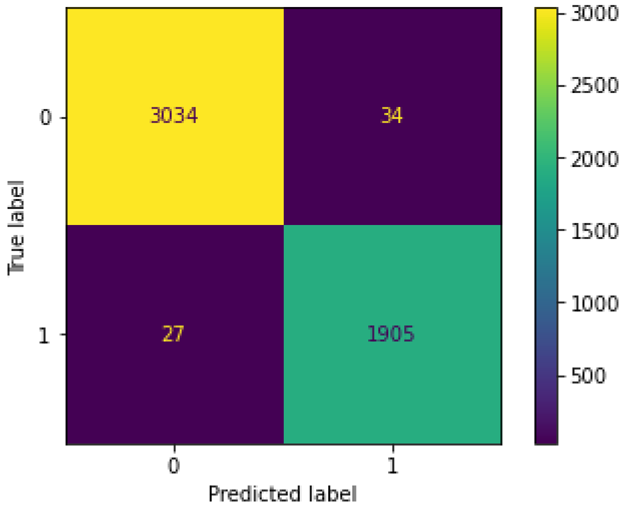
Out[46]:
```
RandomizedSearchCV(cv=10, estimator=LGBMClassifier(), n_jobs=-1,
                   param_distributions={'learning_rate': [0.5, 0.1],
                                        'max_depth': [3, 5, 7],
                                        'n_estimators': [64, 128, 256]},
                   random_state=25, scoring='roc_auc')
```

In [47]:
```
# tuned hyperparameters

print('Best Parameters : {}'.format(randomized_object.best_params_))
print('Best_AUC_score : {}'.format(round(randomized_object.best_score_, 4)))
print('Best model : {}'.format(randomized_object.best_estimator_))
CV_Res = pd.concat([pd.DataFrame(randomized_object.cv_results_['params']),
                    pd.DataFrame(randomized_object.cv_results_['mean_test_score'], columns=['AUC_score'])], axis=1)
CV_Res = CV_Res.sort_values(by='AUC_score', ascending=False)
print(CV_Res)
```

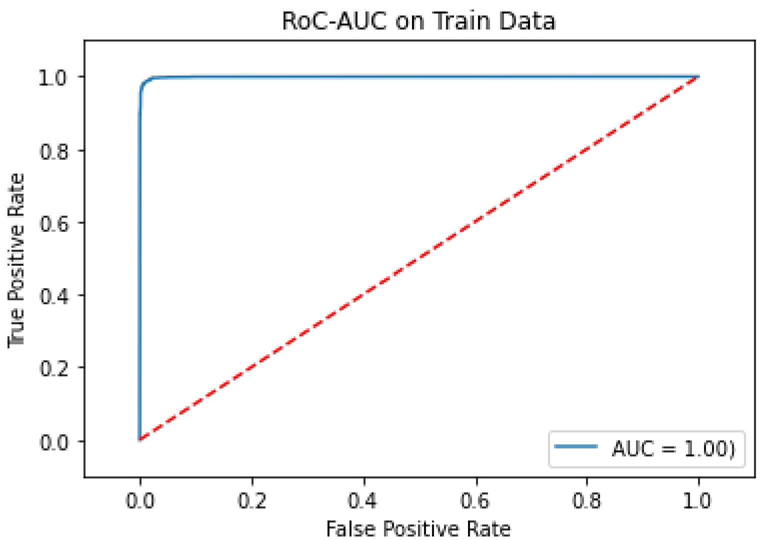
```
Best Parameters : {'n_estimators': 256, 'max_depth': 3, 'learning_rate': 0.1}
Best_AUC_score : 0.9978
Best model : LGBMClassifier(max_depth=3, n_estimators=256)
   n_estimators  max_depth  learning_rate  AUC_score
0           256          3        0.10000    0.99782
4            64          3        0.50000    0.99776
6           128          3        0.50000    0.99744
2           128          3        0.10000    0.99737
3           128          7        0.10000    0.99736
1           256          5        0.10000    0.99726
5            64          5        0.50000    0.99705
9           256          7        0.10000    0.99701
7           256          5        0.50000    0.99636
8            64          3        0.10000    0.99551
```

In [48]:
```
# Tuned model performance on Train and Test data

clf_report(randomized_object.best_estimator_, X_train_clf, Y_train_clf, 'Train')
clf_report(randomized_object.best_estimator_, X_test_clf, Y_test_clf, 'Test')
```

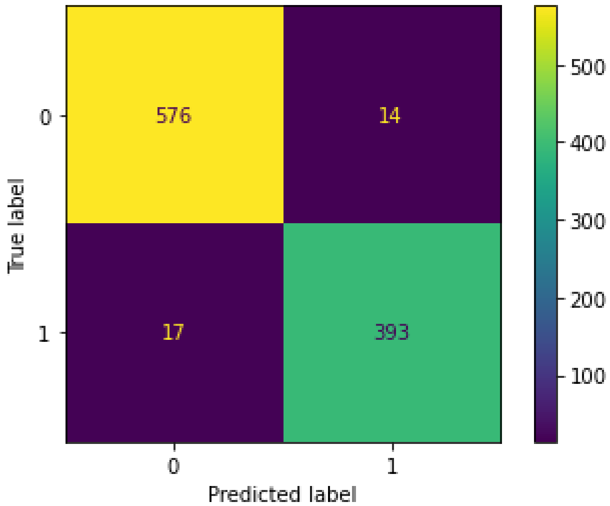
```
Classification report for Train data
```



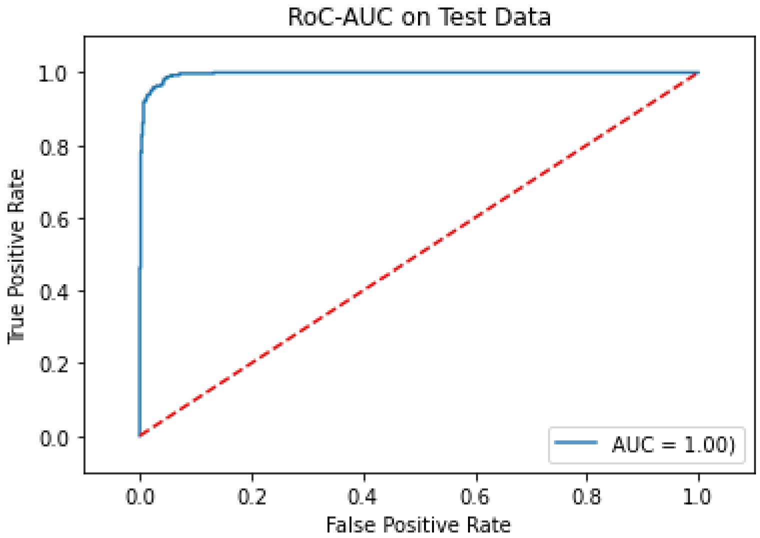

```
Overall Accuracy : 98.78
Precision Score : 98.25
Recall Score : 98.6
AUC : 99.94
```



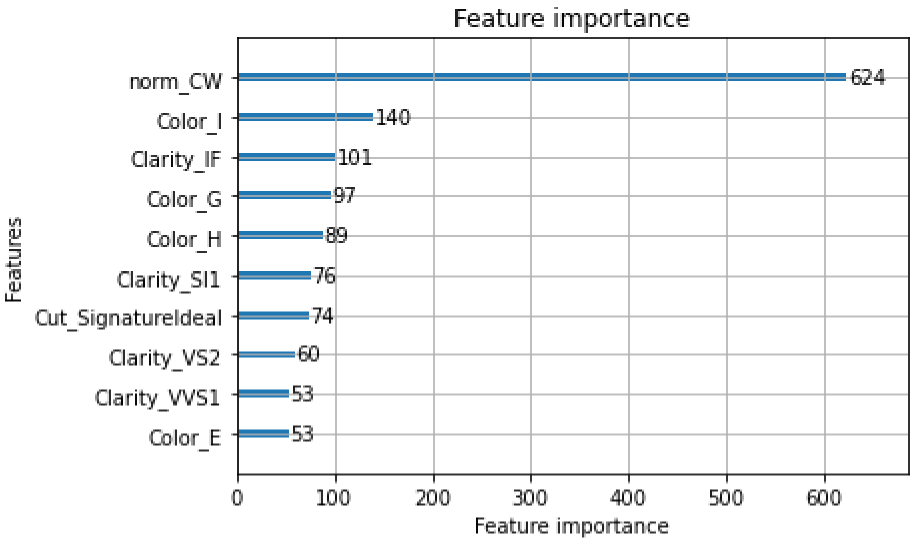

```
Classification report for Test data
```




```
Overall Accuracy : 96.9
Precision Score : 96.56
Recall Score : 95.85
AUC : 99.6
```

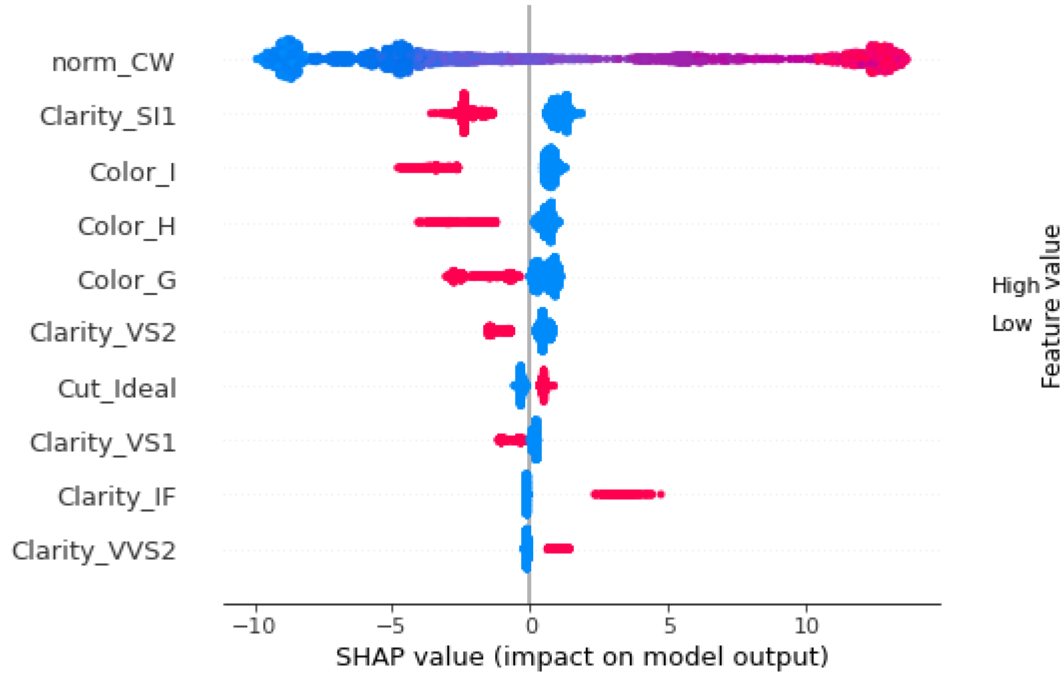


In [49]:
```
lgb.plot_importance(randomized_object.best_estimator_, max_num_features=10)
plt.show()
```



In [50]:

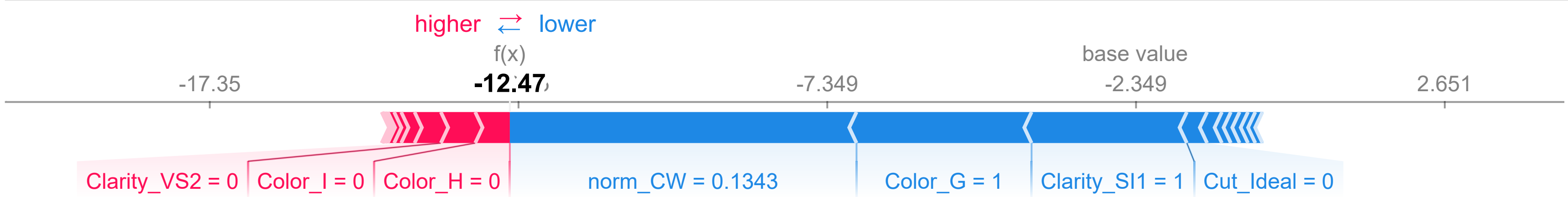
```
explainer = shap.TreeExplainer(randomized_object.best_estimator_)
shap_values = explainer.shap_values(X_train_clf)
shap.summary_plot(shap_values[1], X_train_clf, max_display=10)
plt.show()
```



In [51]:

```
shap.force_plot(explainer.expected_value[1], shap_values[1][0,:], X_train_clf.iloc[0, :])
```

Out[51]:



4. Regression

In [52]:

```
# descriptive statistics for categorical variables

df_reg.describe(include='object')
```

Out[52]:

	Cut	Color	Clarity	Polish	Symmetry	Report

	Cut	Color	Clarity	Polish	Symmetry	Report
count	6000	6000	6000	6000	6000	6000
unique	5	6	7	4	4	2
top	Ideal	G	SI1	EX	VG	GIA
freq	2482	1501	2059	2425	2417	5266

In [53]:

```
# descriptive statistics for numeric variables

df_reg.describe(include='number')
```

Out[53]:

	ID	Carat Weight	Price
count	6000.00000	6000.00000	6000.00000
mean	3000.50000	1.33452	11791.57933
std	1732.19514	0.47570	10184.35005
min	1.00000	0.75000	2184.00000
25%	1500.75000	1.00000	5150.50000
50%	3000.50000	1.13000	7857.00000
75%	4500.25000	1.59000	15036.50000
max	6000.00000	2.91000	101561.00000

4.1 Data Cleaning and Exploratory Analysis

In [54]:

```
# removing white spaces from the column names

dictionary = {' ' : '_', '-' : ''}
df_reg.replace(dictionary, regex=True, inplace=True)
df_reg.columns = df_reg.columns.str.replace(' ', '_')
display(df_reg.head())
display(df_reg.tail())
```

ID	Carat_Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price

	ID	Carat_Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
0	1	1.10000	Ideal	H	SI1	VG	EX	GIA	5169.00000
1	2	0.83000	Ideal	H	VS1	ID	ID	AGSL	3470.00000
2	3	0.85000	Ideal	H	SI1	EX	EX	GIA	3183.00000
3	4	0.91000	Ideal	E	SI1	VG	VG	GIA	4370.00000
4	5	0.83000	Ideal	G	SI1	EX	EX	GIA	3171.00000

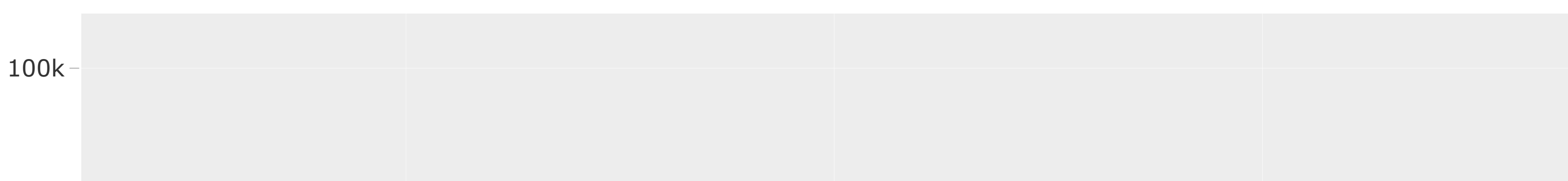
	ID	Carat_Weight	Cut	Color	Clarity	Polish	Symmetry	Report	Price
5995	5996	1.03000	Ideal	D	SI1	EX	EX	GIA	6250.00000
5996	5997	1.00000	Very_Good	D	SI1	VG	VG	GIA	5328.00000
5997	5998	1.02000	Ideal	D	SI1	EX	EX	GIA	6157.00000
5998	5999	1.27000	SignatureIdeal	G	VS1	EX	EX	GIA	11206.00000
5999	6000	2.19000	Ideal	E	VS1	EX	EX	GIA	30507.00000

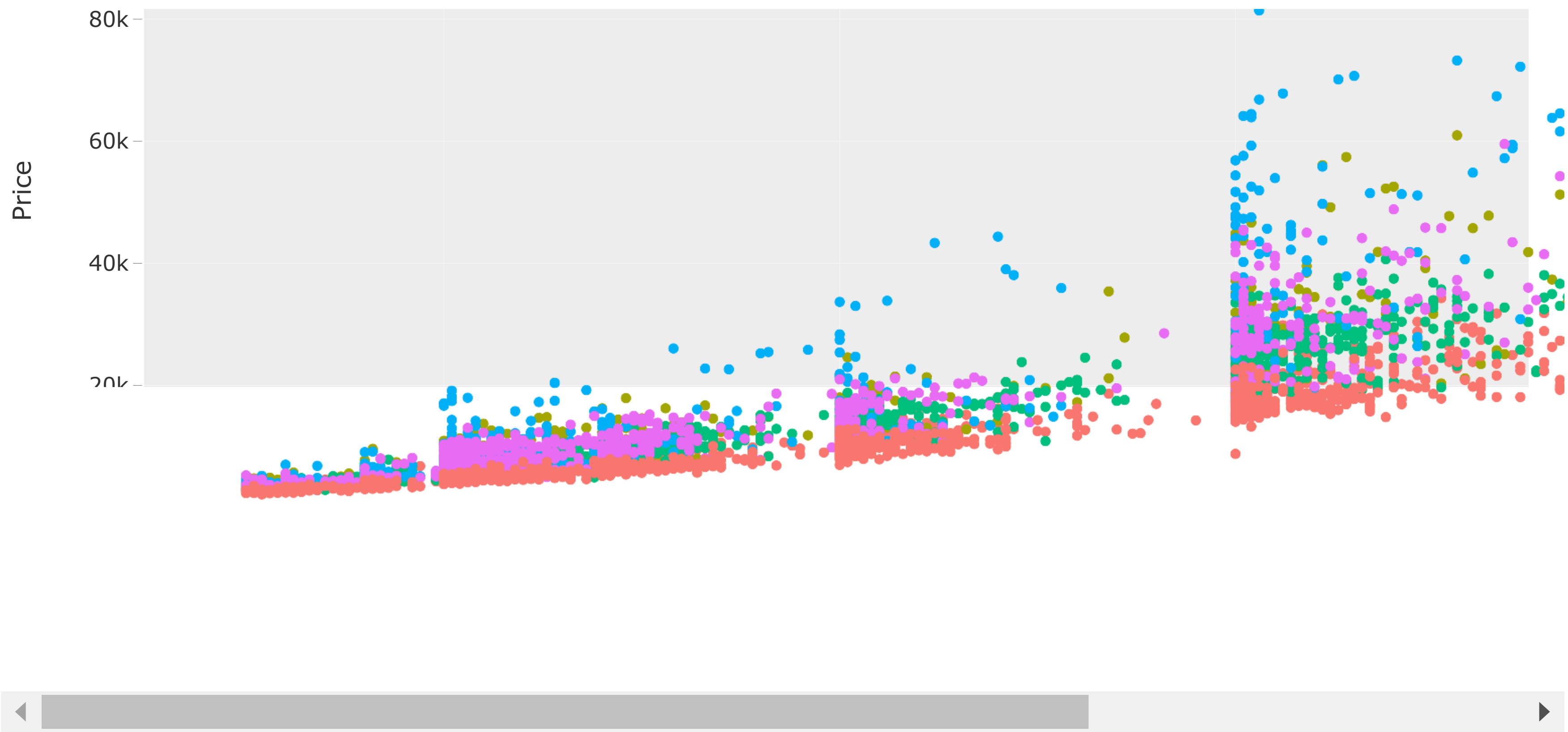
In [55]:

```
# creating new features with some mathematical-transformation

df_reg['log_CW'] = np.log(df_reg['Carat_Weight'])
df_reg['norm_CW'] = MinMaxScaler().fit_transform(df_reg[['Carat_Weight']])
df_reg['lgnm_CW'] = np.log(df_reg['norm_CW']+0.00001)
```

In [56]:

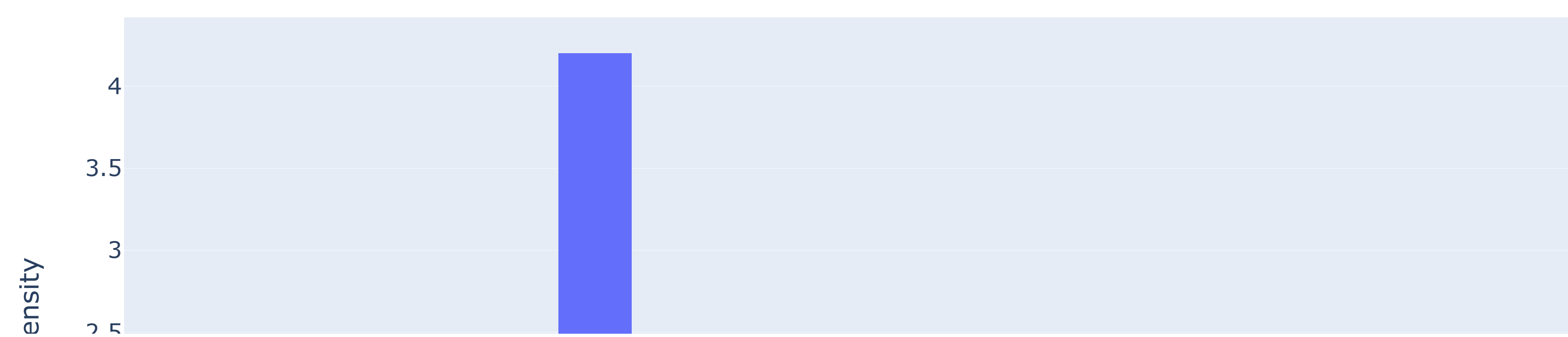
```
# scatter plot for Carat_Weight with respect to different features to observe existing correlations

fig = px.scatter(df_reg, x='Carat_Weight', y='Price', color='Color', template='ggplot2')
fig.show()
```

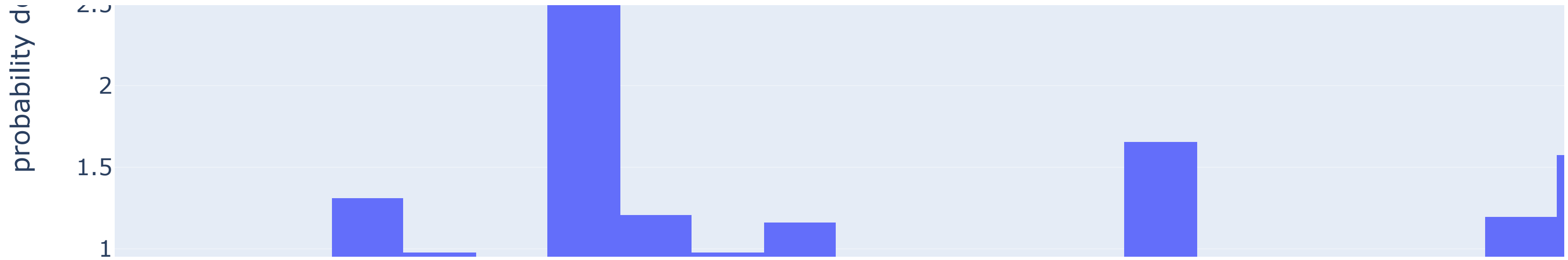




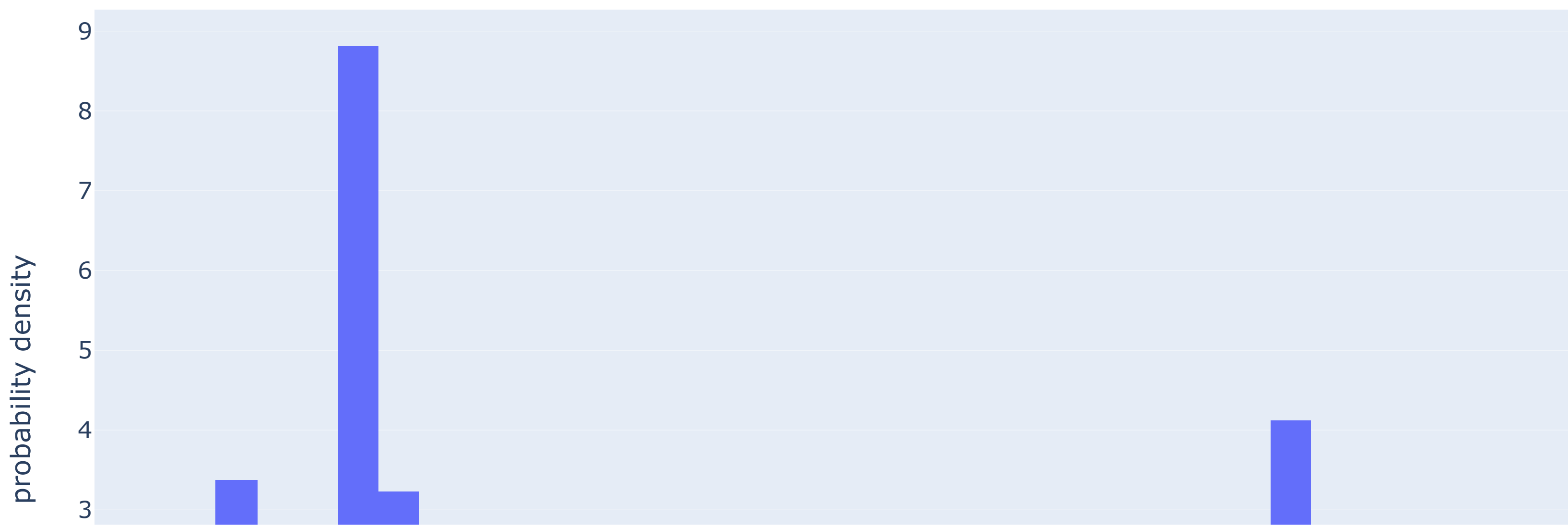
In [57]:

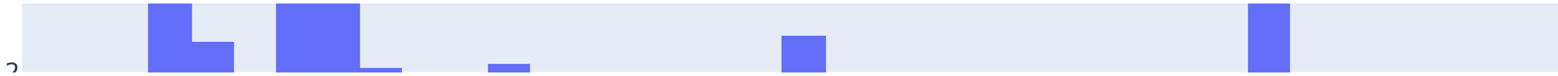
```
# histogram plots to check Log(Carat_Weight) distribution

fig = px.histogram(df_reg, x='log_CW', histnorm='probability density')
fig.show()
```





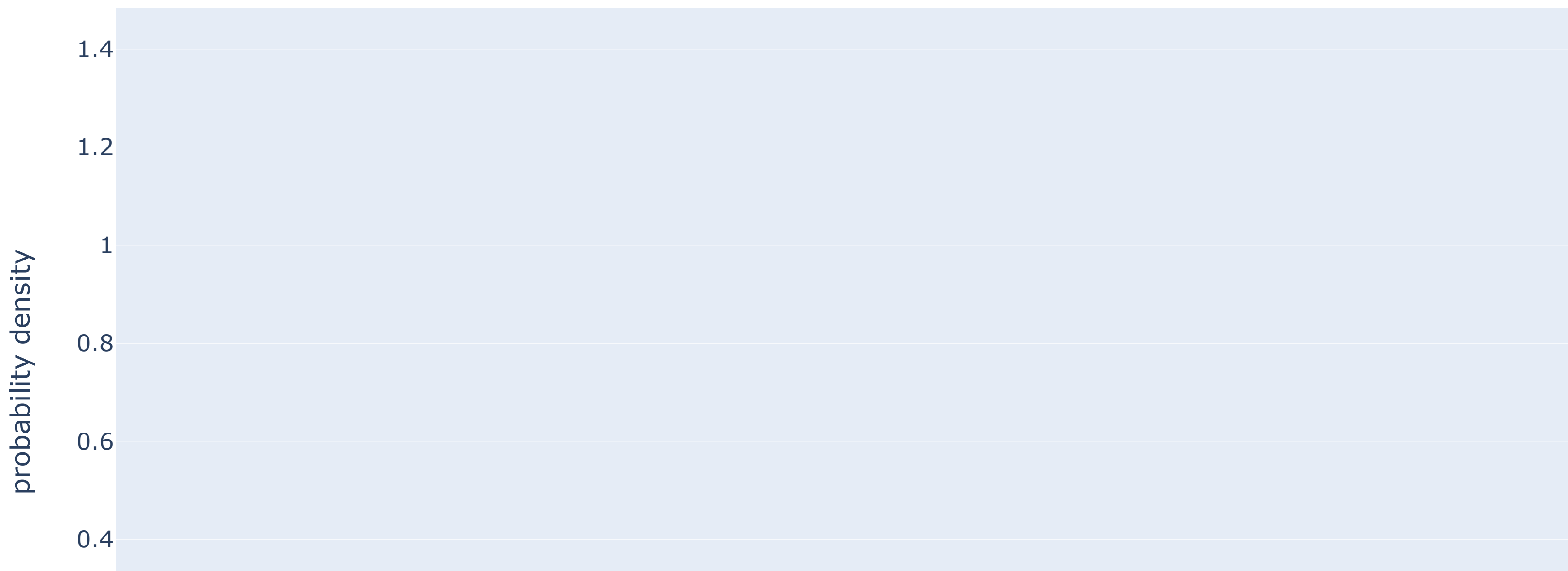
In [58]:
```
# histogram plots to check Normalized(Carat_Weight) distribution

fig = px.histogram(df_reg, x='norm_CW', histnorm='probability density')
fig.show()
```





In [59]:

```
# histogram plots to check Log(Normalized(Carat_Weight)) distribution

fig = px.histogram(df_reg, x='lgnm_CW', histnorm='probability density')
fig.show()
```



4.2 Pre-processing

In [60]:

```
# collecting all the numerical and categorical variables and saving them in two different lists

categorical_columns_seen = [c for i,c in enumerate(df_reg.columns) if df_reg.dtypes[i] in [object]]
numerical_columns_seen = [c for i,c in enumerate(df_reg.columns) if df_reg.dtypes[i] not in [object]]
categorical_columns_seen, numerical_columns_seen
```

Out[60]:

```
(['Cut', 'Color', 'Clarity', 'Polish', 'Symmetry', 'Report'],
 ['ID', 'Carat_Weight', 'Price', 'log_CW', 'norm_CW', 'lgnm_CW'])
```

In [61]:

```
# creating one hot encoded labels for categorical data

encoder = OneHotEncoder(handle_unknown='error', drop='first')
encoded_df = (pd.DataFrame(encoder.fit_transform(df_reg[categorical_columns_seen]).toarray())).astype(int)
encoded_df.columns = encoder.get_feature_names_out(categorical_columns_seen)
encoded_df.head()
```

Out[61]:

	Cut_Good	Cut_Ideal	Cut_SignatureIdeal	Cut_Very_Good	Color_E	Color_F	Color_G	Color_H	Color_I	Clarity_IF	...	Clarity_VS2	Clarity_VVS
0	0	1	0	0	0	0	0	1	0	0	...	0	
1	0	1	0	0	0	0	0	1	0	0	...	0	
2	0	1	0	0	0	0	0	1	0	0	...	0	
3	0	1	0	0	1	0	0	0	0	0	...	0	
4	0	1	0	0	0	0	1	0	0	0	...	0	

5 rows × 22 columns

In [62]:

```
tmp_df = pd.DataFrame()
tmp_df[numerical_columns_seen] = df_reg[numerical_columns_seen]
```

```
cols = list(encoded_df.columns.values)
tmp_df[cols] = encoded_df[cols]
tmp_df.head()
```

Out[62]:

	ID	Carat_Weight	Price	log_CW	norm_CW	lgnm_CW	Cut_Good	Cut_Ideal	Cut_SignatureIdeal	Cut_Very_Good	...	Clarity_VS2	Clari
0	1	1.10000	5169.00000	0.09531	0.16204	-1.81987	0	1	0	0	...	0	
1	2	0.83000	3470.00000	-0.18633	0.03704	-3.29557	0	1	0	0	...	0	
2	3	0.85000	3183.00000	-0.16252	0.04630	-3.07248	0	1	0	0	...	0	
3	4	0.91000	4370.00000	-0.09431	0.07407	-2.60255	0	1	0	0	...	0	
4	5	0.83000	3171.00000	-0.18633	0.03704	-3.29557	0	1	0	0	...	0	

5 rows × 28 columns

4.3 Model Development

In [63]:

```
# creating Train and test data
Y_reg = tmp_df['Price']
X_reg = tmp_df.drop(['ID', 'Price', 'norm_CW', 'lgnm_CW', 'Carat_Weight'], axis = 1)

X_train_reg, X_test_reg, Y_train_reg, Y_test_reg = train_test_split(X_reg, Y_reg, train_size = 0.8334, random_state = 25)
X_train_reg.shape, X_test_reg.shape
```

Out[63]: ((5000, 23), (1000, 23))

In [64]:

```
# model 1 : Light Gradient Boosting model

lgb_model = lgb.LGBMRegressor(boosting_type='goss', num_leaves=16, max_depth=- 1, learning_rate=0.1, n_estimators=64,
                              objective='regression', reg_lambda=0.4, random_state=25, n_jobs=- 1, importance_type='gain'

lgb_model.fit(X_train_reg, Y_train_reg)
Y_pred_reg = lgb_model.predict(X_test_reg)
```

In [65]:

```
# checking model performance on Train and test data
```

```
train_metrics(lgb_model, X_train_reg, Y_train_reg)
test_metrics(Y_test_reg, Y_pred_reg)
```

```
Cross Validated Metric Results for Train Data:
         R-sq     MAE  MAPE
0        0.98  717.63  6.61
1        0.96  824.34  6.46
2        0.98  760.91  6.45
3        0.98  783.78  6.30
4        0.96 1001.80  7.31
5        0.96  820.40  6.44
6        0.96  833.13  5.94
7        0.96  889.92  6.85
8        0.98  792.09  6.68
9        0.97  825.30  6.84
Mean     0.97  824.93  6.59
Std Dev  0.01   73.47  0.35

Metric Results for Test Data:
   R-sq    MAE  MAPE
0  0.95 810.86  6.38
```

4.4 Hyperparameter Tuning

In [66]:

```
params = {'n_estimators' : [64, 128, 256], 'max_depth' : [3,5,7], 'learning_rate' : [0.5, 0.1]}
randomized_object = RandomizedSearchCV(estimator=xgb.XGBRegressor(), scoring=make_scorer(neg_mean_absolute_percentage_err
                                       random_state=25, cv=10, n_jobs=-1, param_distributions=params, verbose=0)
randomized_object.fit(X_train_reg, Y_train_reg)
```

Out[66]:

```
RandomizedSearchCV(cv=10,
                   estimator=XGBRegressor(base_score=None, booster=None,
                                          colsample_bylevel=None,
                                          colsample_bynode=None,
                                          colsample_bytree=None,
                                          enable_categorical=False, gamma=None,
                                          gpu_id=None, importance_type=None,
                                          interaction_constraints=None,
                                          learning_rate=None,
                                          max_delta_step=None, max_depth=None,
                                          min_child_weight=None, missing=nan,
                                          monotone_constraints...
                                          num_parallel_tree=None,
```

```
                                          predictor=None, random_state=None,
                                          reg_alpha=None, reg_lambda=None,
                                          scale_pos_weight=None, subsample=None,
                                          tree_method=None,
                                          validate_parameters=None,
                                          verbosity=None),
                   n_jobs=-1,
                   param_distributions={'learning_rate': [0.5, 0.1],
                                        'max_depth': [3, 5, 7],
                                        'n_estimators': [64, 128, 256]},
                   random_state=25,
                   scoring=make_scorer(neg_mean_absolute_percentage_error))
```

In [67]:

```
# tuned hyperparameters

print('Best Parameters : {}'.format(randomized_object.best_params_))
print('Best_MAPE_score : {}'.format(round(randomized_object.best_score_, 4)))
print('Best model : {}'.format(randomized_object.best_estimator_))
CV_Res = pd.concat([pd.DataFrame(randomized_object.cv_results_['params']),
                    pd.DataFrame(randomized_object.cv_results_['mean_test_score'], columns=['MAPE_score'])], axis=1)
CV_Res = CV_Res.sort_values(by='MAPE_score', ascending=False)
print(CV_Res)
```

```
Best Parameters : {'n_estimators': 128, 'max_depth': 7, 'learning_rate': 0.1}
Best_MAPE_score : -5.2092
Best model : XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
             gamma=0, gpu_id=-1, importance_type=None,
             interaction_constraints='', learning_rate=0.1, max_delta_step=0,
             max_depth=7, min_child_weight=1, missing=nan,
             monotone_constraints='()', n_estimators=128, n_jobs=4,
             num_parallel_tree=1, predictor='auto', random_state=0, reg_alpha=0,
             reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method='exact',
             validate_parameters=1, verbosity=None)
   n_estimators  max_depth  learning_rate  MAPE_score
3           128          7        0.10000    -5.20919
9           256          7        0.10000    -5.24864
1           256          5        0.10000    -5.28899
7           256          5        0.50000    -5.77052
5            64          5        0.50000    -5.82666
6           128          3        0.50000    -6.33404
0           256          3        0.10000    -6.75647
4            64          3        0.50000    -7.18114
```

```
2           128          3        0.10000    -7.53149
8            64          3        0.10000    -9.12262
```

In [68]:

```python
# Tuned model performance on Train and Test data

Y_pred_reg = randomized_object.best_estimator_.predict(X_test_reg)
train_metrics(randomized_object.best_estimator_, X_train_reg, Y_train_reg)
test_metrics(Y_test_reg, Y_pred_reg)
```

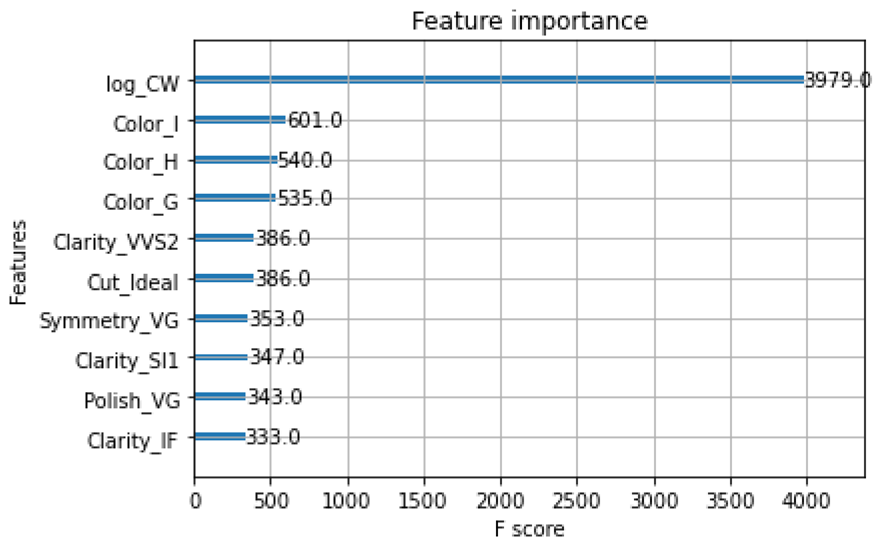
```
Cross Validated Metric Results for Train Data:
         R-sq    MAE  MAPE
0        0.99 613.45  5.14
1        0.98 645.88  5.06
2        0.98 673.20  5.32
3        0.99 693.89  5.19
4        0.97 784.98  5.44
5        0.99 641.57  5.08
6        0.99 649.51  4.90
7        0.97 738.30  5.53
8        0.98 667.26  5.07
9        0.98 686.24  5.14
Mean     0.98 679.43  5.19
Std Dev  0.01  47.81  0.18

Metric Results for Test Data:
   R-sq    MAE  MAPE
0  0.96 674.15  5.00
```

In [69]:

```python
# creating feature importance plot

xgb.plot_importance(randomized_object.best_estimator_, max_num_features=10)
```

Out[69]:

```
<AxesSubplot:title={'center':'Feature importance'}, xlabel='F score', ylabel='Features'>
```



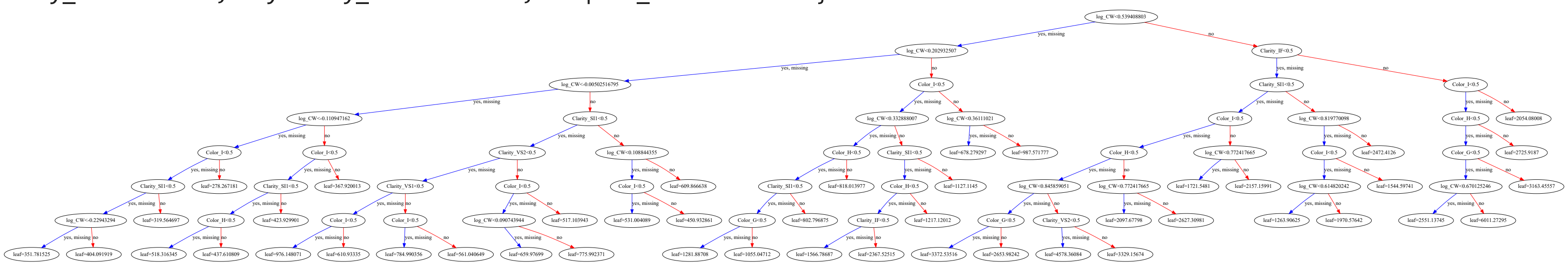
In [70]:

```
# creating features tree plot to explain tuned-model decisions

tuned_params = randomized_object.best_estimator_.get_booster()
for importance_type in ('weight', 'gain', 'cover', 'total_gain', 'total_cover'):
    print('%s: ' % importance_type, tuned_params.get_score(importance_type=importance_type))

xgb.to_graphviz(randomized_object.best_estimator_, numtrees=0, size='20,20')
```

```
weight:  {'log_CW': 3979.0, 'Cut_Good': 251.0, 'Cut_Ideal': 386.0, 'Cut_SignatureIdeal': 186.0, 'Cut_Very_Good': 210.0,
'Color_E': 233.0, 'Color_F': 327.0, 'Color_G': 535.0, 'Color_H': 540.0, 'Color_I': 601.0, 'Clarity_IF': 333.0, 'Clarity_S
I1': 347.0, 'Clarity_VS1': 285.0, 'Clarity_VS2': 326.0, 'Clarity_VVS1': 201.0, 'Clarity_VVS2': 386.0, 'Polish_G': 150.0,
'Polish_ID': 138.0, 'Polish_VG': 343.0, 'Symmetry_G': 200.0, 'Symmetry_ID': 27.0, 'Symmetry_VG': 353.0, 'Report_GIA': 11
1.0}
gain:  {'log_CW': 478909536.0, 'Cut_Good': 4592406.5, 'Cut_Ideal': 11844989.0, 'Cut_SignatureIdeal': 19034940.0, 'Cut_Ver
y_Good': 5173736.0, 'Color_E': 14444548.0, 'Color_F': 21241536.0, 'Color_G': 62449032.0, 'Color_H': 112486464.0, 'Color_
I': 126427296.0, 'Clarity_IF': 430294912.0, 'Clarity_SI1': 229794448.0, 'Clarity_VS1': 61524256.0, 'Clarity_VS2': 7511272
0.0, 'Clarity_VVS1': 116088600.0, 'Clarity_VVS2': 55291200.0, 'Polish_G': 2205670.5, 'Polish_ID': 7328479.0, 'Polish_VG':
3530424.25, 'Symmetry_G': 3260049.0, 'Symmetry_ID': 807730.5, 'Symmetry_VG': 4839952.5, 'Report_GIA': 3286022.25}
cover:  {'log_CW': 452.32244873046875, 'Cut_Good': 180.37449645996094, 'Cut_Ideal': 258.0777282714844, 'Cut_SignatureIdea
l': 699.6236572265625, 'Cut_Very_Good': 259.4571533203125, 'Color_E': 390.1673889160156, 'Color_F': 378.9541320800781, 'C
olor_G': 306.6728820800781, 'Color_H': 361.75, 'Color_I': 300.1247863769531, 'Clarity_IF': 586.2973022460938, 'Clarity_SI
1': 412.7521667480469, 'Clarity_VS1': 166.25613403320312, 'Clarity_VS2': 240.76380920410156, 'Clarity_VVS1': 645.55224609
375, 'Clarity_VVS2': 378.63470458984375, 'Polish_G': 372.0, 'Polish_ID': 56.565216064453125, 'Polish_VG': 207.57434082031
25, 'Symmetry_G': 449.7850036621094, 'Symmetry_ID': 55.407405853271484, 'Symmetry_VG': 257.2776184082031, 'Report_GIA': 5
58.7927856445312}
total_gain:  {'log_CW': 1905581096960.0, 'Cut_Good': 1152694016.0, 'Cut_Ideal': 4572165632.0, 'Cut_SignatureIdeal': 35404
```

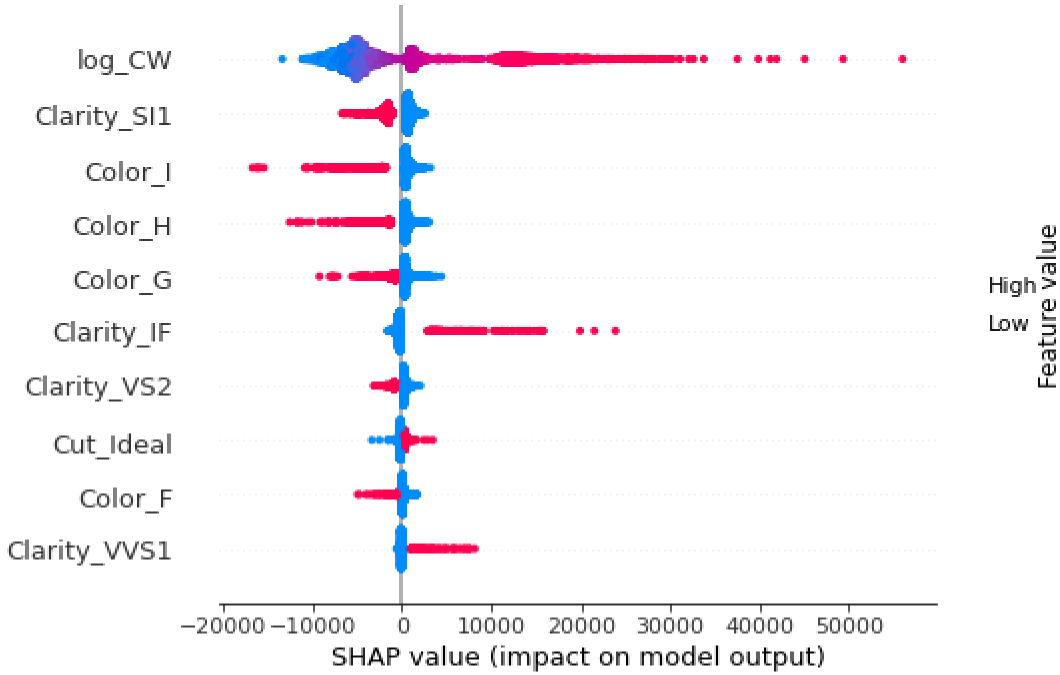
98944.0, 'Cut_Very_Good': 1086484608.0, 'Color_E': 3365579776.0, 'Color_F': 6945981952.0, 'Color_G': 33410232320.0, 'Color_H': 60742688768.0, 'Color_I': 75982807040.0, 'Clarity_IF': 143288205312.0, 'Clarity_SI1': 79738675200.0, 'Clarity_VS1': 17534412800.0, 'Clarity_VS2': 24486746112.0, 'Clarity_VVS1': 23333808128.0, 'Clarity_VVS2': 21342402560.0, 'Polish_G': 330850560.0, 'Polish_ID': 1011330112.0, 'Polish_VG': 1210935552.0, 'Symmetry_G': 652009792.0, 'Symmetry_ID': 21808724.0, 'Symmetry_VG': 1708503168.0, 'Report_GIA': 364748480.0}
total_cover: {'log_CW': 1799791.0, 'Cut_Good': 45274.0, 'Cut_Ideal': 99618.0, 'Cut_SignatureIdeal': 130130.0, 'Cut_Very_Good': 54486.0, 'Color_E': 90909.0, 'Color_F': 123918.0, 'Color_G': 164070.0, 'Color_H': 195345.0, 'Color_I': 180375.0, 'Clarity_IF': 195237.0, 'Clarity_SI1': 143225.0, 'Clarity_VS1': 47383.0, 'Clarity_VS2': 78489.0, 'Clarity_VVS1': 129756.0, 'Clarity_VVS2': 146153.0, 'Polish_G': 55800.0, 'Polish_ID': 7806.0, 'Polish_VG': 71198.0, 'Symmetry_G': 89957.0, 'Symmetry_ID': 1496.0, 'Symmetry_VG': 90819.0, 'Report_GIA': 62026.0}

Out[70]:



In [71]:

```
# creating SHAP-explainer plot for all features

explainer = shap.TreeExplainer(randomized_object.best_estimator_)
shap_values = explainer.shap_values(X_train_reg)
shap.summary_plot(shap_values, X_train_reg, max_display=10)
```



In [72]:

```
# creating SHAP-explainer plot for all features but for a single data row

shap.force_plot(explainer.expected_value, shap_values[0,:], X_train_reg.iloc[0, :])
```

Out[72]: