Experiment

November 29, 2025

1 Appendix A

[1]:
```
import pandas as pd
import pycaret
import numpy as np
import copy
```

1.1 Read Raw Data

[2]:
```
df_raw = pd.read_csv('training_data_ht2025.csv')
```

[3]:
```
df_raw
```

[3]:
```
      hour_of_day  day_of_week  month  holiday  weekday  summertime  temp  \
0               5            5      1        0        0           0  -7.2
1              21            4      1        0        1           0  -1.3
2              21            3      8        0        1           1  26.9
3               1            6      1        0        0           0   3.1
4              17            0      3        0        1           0  11.7
…             …           …     …      …       …        …   …
1595            3            5      6        0        0           1  21.5
1596           14            0      6        0        1           1  23.2
1597           13            0      3        0        1           1  13.9
1598           14            5      3        0        0           1  11.7
1599           22            6      2        0        0           0   4.2

       dew  humidity  precip  snow  snowdepth  windspeed  cloudcover  \
0    -15.0     53.68   0.000     0        0.0       16.3        31.6
1    -12.8     40.97   0.000     0        0.0       23.9        85.7
2     21.8     73.39   0.000     0        0.0        0.0        81.1
3     -4.0     59.74   0.000     0        0.0       19.2         0.0
4    -11.4     18.71   0.000     0        0.0       10.5        44.6
…     …       …      …     …       …         …
1595  19.4     87.68   0.000     0        0.0       10.6        24.4
1596  20.1     82.43   2.217     0        0.0        9.8        92.1
1597  -2.2     32.93   0.000     0        2.0       18.2        79.3
1598  -9.3     22.09   0.000     0        0.0        5.8        24.4
```

```
1599   1.7     84.11   1.081     0         0.0       21.9         97.4

      visibility    increase_stock
0           16.0   low_bike_demand
1           16.0   low_bike_demand
2           16.0   low_bike_demand
3           16.0   low_bike_demand
4           16.0   low_bike_demand
…          …              …
1595        16.0   low_bike_demand
1596        10.4   low_bike_demand
1597        16.0   low_bike_demand
1598        16.0  high_bike_demand
1599        16.0   low_bike_demand

[1600 rows x 16 columns]
```

1.1.1 Check Data Contents

[4]:
```
df_raw['increase_stock'].value_counts()
```

[4]:
```
increase_stock
low_bike_demand     1312
high_bike_demand     288
Name: count, dtype: int64
```

Based on above code that shows the values on the target variable *increase_stock*, there's a problem of Imbalance Dataset, where the records with low bike demand are much more compared to records with high bike demands.

[5]:
```
df_raw['holiday'].value_counts()
```

[5]:
```
holiday
0    1547
1      53
Name: count, dtype: int64
```

[6]:
```
df_raw['weekday'].value_counts()
```

[6]:
```
weekday
1    1136
0     464
Name: count, dtype: int64
```

[7]:
```
df_raw['summertime'].value_counts()
```

[7]:
```
summertime
1    1030
```

```
0     570
Name: count, dtype: int64
```

[8]:
```
df_raw['increase_stock'].value_counts()
```

[8]:
```
increase_stock
low_bike_demand     1312
high_bike_demand     288
Name: count, dtype: int64
```

[9]:
```
df_raw['hour_of_day'].value_counts()
```

[9]:
```
hour_of_day
0     84
16    83
13    78
7     77
23    77
17    72
21    71
14    70
3     69
4     69
5     67
1     67
8     66
12    66
15    64
10    63
9     63
6     62
11    58
19    57
22    56
20    55
18    53
2     53
Name: count, dtype: int64
```

[10]:
```
df_raw['day_of_week'].value_counts()
```

[10]:
```
day_of_week
4    242
0    240
5    233
6    231
2    223
```

```
3    220
1    211
Name: count, dtype: int64
```

[11]:
```
df_raw['month'].value_counts()
```

[11]:
```
month
3     147
11    141
4     140
6     137
7     136
12    136
5     133
9     131
2     131
1     128
8     121
10    119
Name: count, dtype: int64
```

[12]:
```
df_raw['snow'].value_counts()
```

[12]:
```
snow
0    1600
Name: count, dtype: int64
```

Note: The Snow Features only have one value: 0, and therefore isn't really useful for this dataset. And In Feature selection we can drop them.

[13]:
```
df_raw.isnull().sum()
```

[13]:
```
hour_of_day      0
day_of_week      0
month            0
holiday          0
weekday          0
summertime       0
temp             0
dew              0
humidity         0
precip           0
snow             0
snowdepth        0
windspeed        0
cloudcover       0
visibility       0
increase_stock   0
```

dtype: int64

Based on the checking results, there're no NaN values, so we can proceed with feature processing.

Cyclical Encoding Function to encode Ordinal Variables into Cyclical encoding using Sine and Cosine

[14]:
```
def cyclical_encode(x, max_val, start_val=0):
    x_arr = np.asarray(x, dtype=float)

    # If values are 1..period (e.g. months 1..12), shift to 0..period-1
    if not start_val:
        x_arr = x_arr - 1

    angle = 2 * np.pi * x_arr / max_val
    sin_x = np.sin(angle)
    cos_x = np.cos(angle)

    if np.isscalar(x):
        return float(sin_x), float(cos_x)

    return sin_x, cos_x
```

[ ]:
```

```

[15]:
```
# Transform Cyclical Values for hour_of_day

# Transform Cyclical Values for hour_of_day
# Hour of day ranges from 0 to 23
df_hour_feat = pd.DataFrame(df_raw['hour_of_day'].apply(lambda x:␣
↪cyclical_encode(x, max(df_raw['hour_of_day']), min(df_raw['hour_of_day']))).
↪to_list())
df_hour_feat.columns = ['hour_of_day_sin', 'hour_of_day_cos']
df_hour_feat


# Transform Cyclical Values for day_of_week
# Day of week ranges from 0 (Monday) to 6 (Sunday)
df_day_feat = pd.DataFrame(df_raw['day_of_week'].apply(lambda x:␣
↪cyclical_encode(x, 6, 0)).to_list())
df_day_feat.columns = ['day_of_week_sin', 'day_of_week_cos']
df_day_feat

# Transform Cyclical Values for month
# Month ranges from 1 to 12
df_month_feat = pd.DataFrame(df_raw['month'].apply(lambda x: cyclical_encode(x,␣
↪12, 1)).to_list())
```

```
df_month_feat.columns = ['month_sin', 'month_cos']
df_month_feat

# Concat Cyclical Features
df_cyclical = pd.concat([df_hour_feat,df_day_feat, df_month_feat], axis=1)
df_cyclical
```

[15]:

```
      hour_of_day_sin  hour_of_day_cos  day_of_week_sin  day_of_week_cos  \
0            0.887885         0.460065    -8.660254e-01             -0.5
1           -0.730836         0.682553     1.224647e-16             -1.0
2           -0.730836         0.682553     8.660254e-01             -0.5
3            0.000000         1.000000    -8.660254e-01              0.5
4           -0.942261        -0.334880    -8.660254e-01              0.5
…                 …               …                …               …
1595         0.519584         0.854419    -8.660254e-01             -0.5
1596        -0.398401        -0.917211    -8.660254e-01              0.5
1597        -0.136167        -0.990686    -8.660254e-01              0.5
1598        -0.398401        -0.917211    -8.660254e-01             -0.5
1599        -0.519584         0.854419    -8.660254e-01              0.5

         month_sin     month_cos
0     5.000000e-01  8.660254e-01
1     5.000000e-01  8.660254e-01
2    -8.660254e-01 -5.000000e-01
3     5.000000e-01  8.660254e-01
4     1.000000e+00  6.123234e-17
…              …             …
1595  1.224647e-16 -1.000000e+00
1596  1.224647e-16 -1.000000e+00
1597  1.000000e+00  6.123234e-17
1598  1.000000e+00  6.123234e-17
1599  8.660254e-01  5.000000e-01

[1600 rows x 6 columns]
```

[16]:

```
# Creating Final Feature and Target DataFrames
df_features = copy.deepcopy(df_raw)

# Creating Target Variable and Mapping text to binary
df_target = df_features['increase_stock'].map({'low_bike_demand': 0,␣
↪'high_bike_demand': 1})

# Dropping original cyclical columns and target from features
df_features = df_features.drop(['hour_of_day', 'day_of_week', 'month',␣
↪'increase_stock'], axis=1)

# Concatenating Cyclical Features to Features DataFrame
```

```
df_features = pd.concat([df_cyclical, df_features], axis=1)

df_features
```

[16]:

```
      hour_of_day_sin  hour_of_day_cos  day_of_week_sin  day_of_week_cos  \
0            0.887885         0.460065    -8.660254e-01             -0.5
1           -0.730836         0.682553     1.224647e-16             -1.0
2           -0.730836         0.682553     8.660254e-01             -0.5
3            0.000000         1.000000    -8.660254e-01              0.5
4           -0.942261        -0.334880    -8.660254e-01              0.5
…                 …              …                …                …
1595         0.519584         0.854419    -8.660254e-01             -0.5
1596        -0.398401        -0.917211    -8.660254e-01              0.5
1597        -0.136167        -0.990686    -8.660254e-01              0.5
1598        -0.398401        -0.917211    -8.660254e-01             -0.5
1599        -0.519584         0.854419    -8.660254e-01              0.5

         month_sin     month_cos  holiday  weekday  summertime  temp   dew  \
0     5.000000e-01  8.660254e-01        0        0           0  -7.2 -15.0
1     5.000000e-01  8.660254e-01        0        1           0  -1.3 -12.8
2    -8.660254e-01 -5.000000e-01        0        1           1  26.9  21.8
3     5.000000e-01  8.660254e-01        0        0           0   3.1  -4.0
4     1.000000e+00  6.123234e-17        0        1           0  11.7 -11.4
…              …            …       …        …          …   …   …
1595  1.224647e-16 -1.000000e+00        0        0           1  21.5  19.4
1596  1.224647e-16 -1.000000e+00        0        1           1  23.2  20.1
1597  1.000000e+00  6.123234e-17        0        1           1  13.9  -2.2
1598  1.000000e+00  6.123234e-17        0        0           1  11.7  -9.3
1599  8.660254e-01  5.000000e-01        0        0           0   4.2   1.7

      humidity  precip  snow  snowdepth  windspeed  cloudcover  visibility
0        53.68   0.000     0        0.0       16.3        31.6        16.0
1        40.97   0.000     0        0.0       23.9        85.7        16.0
2        73.39   0.000     0        0.0        0.0        81.1        16.0
3        59.74   0.000     0        0.0       19.2         0.0        16.0
4        18.71   0.000     0        0.0       10.5        44.6        16.0
…          …     …    …       …          …          …
1595     87.68   0.000     0        0.0       10.6        24.4        16.0
1596     82.43   2.217     0        0.0        9.8        92.1        10.4
1597     32.93   0.000     0        2.0       18.2        79.3        16.0
1598     22.09   0.000     0        0.0        5.8        24.4        16.0
1599     84.11   1.081     0        0.0       21.9        97.4        16.0

[1600 rows x 18 columns]
```

[17]:

```
df_target
```

[17]:
```
0       0
1       0
2       0
3       0
4       0
       ..
1595    0
1596    0
1597    0
1598    1
1599    0
Name: increase_stock, Length: 1600, dtype: int64
```

[18]:
```
# Correlation Analysis between Features and Target
df_corr = pd.concat([df_features, df_target], axis=1).corr()
df_corr['increase_stock'].sort_values(key=abs, ascending=False)
```

[18]:
```
increase_stock     1.000000
hour_of_day_cos   -0.339960
temp               0.336981
humidity          -0.308726
hour_of_day_sin   -0.308121
summertime         0.216052
month_cos         -0.169059
dew                0.132663
weekday           -0.116446
visibility         0.113443
windspeed          0.096011
month_sin         -0.092078
day_of_week_sin   -0.088152
precip            -0.059304
snowdepth         -0.047526
cloudcover        -0.045534
day_of_week_cos   -0.031473
holiday           -0.004909
snow                    NaN
Name: increase_stock, dtype: float64
```

Based on above result, The variables hour_of_day_cos, temp, humidity, hour_of_day_sin, and summertime held the biggest correlation to the target variable. Because of this, we could check the trends more based on those attributes

[19]:
```
from imblearn.over_sampling import SMOTE

# Applying SMOTE to Balance the Dataset
smote = SMOTE(sampling_strategy='minority')

print("Before SMOTE:\n", df_target.value_counts())
```

```
print()

X_resampled, y_resampled = smote.fit_resample(df_features, df_target)

print("After SMOTE:\n", y_resampled.value_counts())

# Note: Don't use SMOTE outside training data to avoid data leakage, this block␣
↪is just for experimentation purposes. In practice, SMOTE should only be␣
↪applied to the training set within each cross-validation folds.
```

```
Before SMOTE:
 increase_stock
0    1312
1     288
Name: count, dtype: int64

After SMOTE:
 increase_stock
0    1312
1    1312
Name: count, dtype: int64
```

1.2 Model Experiment

[20]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, roc_auc_score, classification_report
from sklearn.model_selection import KFold, train_test_split
from sklearn.model_selection import GridSearchCV
```

K-Fold Classification Report

[21]:
```
## Function to do K-Fold Cross Validation and return the classification report
def kfold_classification_reports(clf, X, y, n_splits=10, shuffle=True,␣
↪random_state=42):
    kf = KFold(n_splits=n_splits, shuffle=shuffle, random_state=random_state)
    reports = []

    for train_index, test_index in kf.split(X):
        # Applying SMOTE to Balance the Dataset
        smote = SMOTE(sampling_strategy='minority')


        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

```
        clf.fit(X_resampled, y_resampled)
        y_pred = clf.predict(X_test)

        report = classification_report(y_test, y_pred, output_dict=True)
        reports += [report]

    detailed = []
    for i, rep in enumerate(reports):
        fold_result = {
            "fold": i + 1,
            "accuracy": rep["accuracy"],
            "precision": rep["weighted avg"]["precision"],
            "recall": rep["weighted avg"]["recall"],
            "f1": rep["weighted avg"]["f1-score"],
        }
        detailed.append(fold_result)

    # compute averaged (generalized) metrics
    accuracies = [d["accuracy"] for d in detailed]
    precisions = [d["precision"] for d in detailed]
    recalls = [d["recall"] for d in detailed]
    f1s = [d["f1"] for d in detailed]

    report_dict = {
        "general": {
            "accuracy_mean": float(np.mean(accuracies)),
            "accuracy_std": float(np.std(accuracies)),
            "precision_mean": float(np.mean(precisions)),
            "precision_std": float(np.std(precisions)),
            "recall_mean": float(np.mean(recalls)),
            "recall_std": float(np.std(recalls)),
            "f1_mean": float(np.mean(f1s)),
            "f1_std": float(np.std(f1s)),
        },
        "detailed": detailed
    }

    return report_dict
```

1.3 Benchmark Model

For the benchmark model, we use a naive model that predict each instance as the majority class in the training dataset. This will provide a baseline accuracy to compare the performance of more sophisticated models.

[22]:
```
from sklearn.dummy import DummyClassifier
```

```
clf_dummy = DummyClassifier(strategy="stratified")
```

[23]:
```
reports = kfold_classification_reports(clf_dummy, df_features, df_target)
reports['general']
```

[23]:
```
{'accuracy_mean': 0.47625,
 'accuracy_std': 0.030593095626301036,
 'precision_mean': 0.6952224635705411,
 'precision_std': 0.032953301949774735,
 'recall_mean': 0.47625,
 'recall_std': 0.030593095626301036,
 'f1_mean': 0.5351422005856692,
 'f1_std': 0.028627093758612 07}
```

1.4 Logistic Regression

For Logistic Regression, we will do hyperparameter tuning on Regularization Strength (C), Penalty type ($penalty$), and Solver type ($solver$)

[ ]:
```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()

#Grid Search to find the best hyperparameters for Logistic Regression
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'C': [0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear', 'lbfgs', 'saga', 'newton-cg']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_lg = grid_search.fit(df_features, df_target)
```

[249]:
```
clf_grid_lg.best_params_
```

[249]:
```
{'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
```

[250]:
```
clf_lg = LogisticRegression(**clf_grid_lg.best_params_)
```

[251]:
```
reports = kfold_classification_reports(clf_lg, df_features, df_target)
reports['general']
```

[251]: {'accuracy_mean': 0.8099999999999999,
 'accuracy_std': 0.026545950726994134,
 'precision_mean': 0.8695895287253285,
 'precision_std': 0.016029365215753214,
 'recall_mean': 0.8099999999999999,
 'recall_std': 0.026545950726994134,
 'f1_mean': 0.8271489924908391,
 'f1_std': 0.02043318475431806}

1.5 Linear Discriminant Analysis (LDA)

For LDA, we us Hyperparameter tuning for Solver Type (*solver*)

[252]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

clf = LinearDiscriminantAnalysis()

# Grid Search to find the best hyperparameters for LDA
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'solver': ['svd', 'lsqr', 'eigen']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_lda = grid_search.fit(df_features, df_target)
```

Fitting 10 folds for each of 3 candidates, totalling 30 fits

[253]:
```
clf_grid_lda.best_params_
```

[253]: {'solver': 'svd'}

[254]:
```
clf_lda = LinearDiscriminantAnalysis(**clf_grid_lda.best_params_)
clf_lda
```

[254]: LinearDiscriminantAnalysis()

[255]:
```
reports = kfold_classification_reports(clf_lda, df_features, df_target)
reports['general']
```

[255]: {'accuracy_mean': 0.795625,
 'accuracy_std': 0.0271065605527518,
 'precision_mean': 0.8675792283319778,
 'precision_std': 0.013589091958582378,

```
 'recall_mean': 0.795625,
 'recall_std': 0.0271065605527518,
 'f1_mean': 0.8155870764945796,
 'f1_std': 0.020417818352304083}
```

1.6 K Nearest Neighbor (KNN)

For K Nearest Neighbor, we use Hyperparameter tuning for selecting the numer of neighbors (*n_neighbors*), weight function used in prediction (*weights*), and the metric used for distance computation (*metric*)

[256]:
```python
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier()

#Grid Search to find the best hyperparameters for KNN
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'n_neighbors': range(3, 21, 2),
        'weights': ['uniform', 'distance'],
        'metric': ['euclidean', 'manhattan', 'minkowski']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_knn = grid_search.fit(df_features, df_target)
```

```
Fitting 10 folds for each of 54 candidates, totalling 540 fits
```

[257]:
```python
clf_grid_knn.best_params_
```

[257]:
```
{'metric': 'manhattan', 'n_neighbors': 3, 'weights': 'distance'}
```

[258]:
```python
clf_knn = KNeighborsClassifier(**clf_grid_knn.best_params_)
clf_knn
```

[258]:
```
KNeighborsClassifier(metric='manhattan', n_neighbors=3, weights='distance')
```

[259]:
```python
reports = kfold_classification_reports(clf_knn, df_features, df_target)
reports['general']
```

[259]:
```
{'accuracy_mean': 0.7675,
 'accuracy_std': 0.020116846174288848,
 'precision_mean': 0.823321421215026,
 'precision_std': 0.026099143909350715,
```

```
 'recall_mean': 0.7675,
 'recall_std': 0.020116846174288848,
 'f1_mean': 0.7863382939494521,
 'f1_std': 0.020420617255737072}
```

1.7 Random Forest Classifier

For Random Forest Classifier, we use Hyperparameter tuning for selecting number of trees (*n_estimator*), maximum depth of each trees (*max_depth*), minimum number of samples required to split (*min_samples_split*), and minimum samples required to be a leaf node (*min_samples_leaf*)

[260]:
```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier()
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'n_estimators': [20, 50, 100, 200, 250],
        'max_depth': [3, 5, 8, 10, 15, 20],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4],
        'criterion': ['gini', 'entropy']
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_rf = grid_search.fit(df_features, df_target)
```

```
Fitting 10 folds for each of 540 candidates, totalling 5400 fits
```

[261]:
```
clf_grid_rf.best_params_
```

[261]:
```
{'criterion': 'entropy',
 'max_depth': 15,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'n_estimators': 250}
```

[262]:
```
clf_rf = RandomForestClassifier(**clf_grid_rf.best_params_)
clf_rf
```

[262]:
```
RandomForestClassifier(criterion='entropy', max_depth=15, n_estimators=250)
```

[263]:
```
reports = kfold_classification_reports(clf_rf, df_features, df_target)
reports['general']
```

[263]: {'accuracy_mean': 0.9037500000000002,
 'accuracy_std': 0.02186606960566987,
 'precision_mean': 0.9065025661469864,
 'precision_std': 0.019154309103939216,
 'recall_mean': 0.9037500000000002,
 'recall_std': 0.02186606960566987,
 'f1_mean': 0.9043197968362898,
 'f1_std': 0.02048728351994524}

1.8 Gradient Boosting

For Gradient Boosting, we use Hyperparameter tuning to determine, the number of sequential estimator (*n_estimators*), the learning rate (*learning_rate*), Maximum depth of the individual regression estimators (*max_depth*), and the fraction of samples to be used for fitting the individual estimators (*subsample*).

[264]:
```
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier()
grid_search = GridSearchCV(
    estimator=clf,
    param_grid={
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.5],
        'max_depth': [3, 5, 8, 10],
        'subsample': [0.6, 0.8, 1.0]
    },
    scoring='f1',
    cv=10,
    n_jobs=-1,
    verbose=1
)
clf_grid_gb = grid_search.fit(df_features, df_target)
```

Fitting 10 folds for each of 180 candidates, totalling 1800 fits

[267]:
```
clf_grid_gb.best_params_
```

[267]: {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 100, 'subsample': 0.6}

[268]:
```
clf_gb = GradientBoostingClassifier(**clf_grid_gb.best_params_)
clf_gb
```

[268]: GradientBoostingClassifier(max_depth=10, subsample=0.6)

[269]:
```
reports = kfold_classification_reports(clf_gb, df_features, df_target)
reports['general']
```

[269]: {'accuracy_mean': 0.899375,
 'accuracy_std': 0.02043472840533977,
 'precision_mean': 0.9014960071645403,
 'precision_std': 0.018078538997051353,
 'recall_mean': 0.899375,
 'recall_std': 0.02043472840533977,
 'f1_mean': 0.8998663647708327,
 'f1_std': 0.01972180961714667}

1.8.1 Saving Model

Saving all model so we can use it later

[270]:
```
# Applying SMOTE to Balance the Full Training Dataset
smote = SMOTE(sampling_strategy='minority')

print("Before SMOTE:\n", df_target.value_counts())

print()

X_resampled, y_resampled = smote.fit_resample(df_features, df_target)

print("After SMOTE:\n", y_resampled.value_counts())
```

```
Before SMOTE:
 increase_stock
0    1312
1     288
Name: count, dtype: int64

After SMOTE:
 increase_stock
0    1312
1    1312
Name: count, dtype: int64
```

[271]:
```
import pickle

# Save the Dummy Classifier model used as benchmark
with open('clf_dummy.pkl', 'wb') as f:
    pickle.dump(clf_dummy, f)
```

[272]:
```
# Train the Logistic Regression model on the full training dataset
clf_lg = LogisticRegression(**clf_grid_lg.best_params_)
clf_lg.fit(X_resampled, y_resampled)

# Save the trained Logistic Regression model
with open('clf_lg.pkl', 'wb') as f:
```

```
    pickle.dump(clf_lg, f)
```

[ ]:
```
# Train the LDA model on the full training dataset
clf_lda = LinearDiscriminantAnalysis(**clf_grid_lda.best_params_)
clf_lda.fit(X_resampled, y_resampled)


# Save the trained LDA model
with open('clf_lda.pkl', 'wb') as f:
    pickle.dump(clf_lda, f)
```

[274]:
```
# Train the KNN model on the full training dataset
clf_knn = KNeighborsClassifier(**clf_grid_knn.best_params_)
clf_knn.fit(X_resampled, y_resampled)

# Save the trained KNN model
with open('clf_knn.pkl', 'wb') as f:
    pickle.dump(clf_knn, f)
```

[275]:
```
# Train the Random Forest Classifier model on the full training dataset
clf_rf = RandomForestClassifier(**clf_grid_rf.best_params_)
clf_rf.fit(X_resampled, y_resampled)

# Save the trained Random Forest Classifier model
with open('clf_rf.pkl', 'wb') as f:
    pickle.dump(clf_rf, f)
```

[276]:
```
# Train the Gradient Boosting Classifier model on the full training dataset
clf_gb = GradientBoostingClassifier(**clf_grid_gb.best_params_)
clf_gb.fit(X_resampled, y_resampled)

# Save the trained Gradient Boosting Classifier model
with open('clf_gb.pkl', 'wb') as f:
    pickle.dump(clf_gb, f)
```

[ ]:
```

```