InsuranceDatasetML

September 24, 2024

1 Import Libraries

[1]:
```
import numpy as np
import pandas as pd
import zipfile
import os
import warnings
warnings.filterwarnings("ignore")
pd.options.mode.copy_on_write = True
```

[2]:
```
import plotly.express as px
import plotly.graph_objects as go
import plotly.io as pio
from plotly.subplots import make_subplots
import plotly.figure_factory as ff

from matplotlib import cm
import matplotlib.pyplot as plt
pio.templates.default = 'plotly_dark'
```

[3]:
```
from sklearn.model_selection import train_test_split,StratifiedKFold
from sklearn.preprocessing import StandardScaler, OneHotEncoder ,OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from scipy import stats
from lightgbm import LGBMClassifier
from sklearn.svm import LinearSVC
from sklearn.linear_model import RidgeClassifierCV
from sklearn.ensemble import StackingClassifier

import optuna
import logging

logging.getLogger('optuna').setLevel(logging.WARNING)
```

1.1 Read Data

[4]:
```
def unzip_file_to_same_location(zip_path):

    extract_to = os.path.dirname(zip_path)

    with zipfile.ZipFile(zip_path, 'r') as zip_ref:

        zip_ref.extractall(extract_to)
        print('Extraction Complete')
```

[5]:
```
zip_file_path = 'playground-series-s4e7.zip'
unzip_file_to_same_location(zip_file_path)
```

```
Extraction Complete
```

[6]:
```
df_train = pd.read_csv('train.csv')
df_test = pd.read_csv('test.csv')
```

[7]:
```
df_train.drop('id',axis=1,inplace=True)
df_test.drop('id',axis=1,inplace=True)
```

[8]:
```
response_counts = df_train['Response'].value_counts(normalize=True)

response_colors = {0: '#4fff51', 1: '#ff0c0c'}

fig = go.Figure()

for response_value in response_counts.index:
    fig.add_trace(
        go.Bar(
            x=[response_value],
            y=[response_counts[response_value]],
            marker_color=response_colors[response_value],

        )
    )

fig.update_layout(
    title='Normalised Count Plot of Response Variable in Training Data',
    xaxis_title='Response',
    yaxis_title='Count',
    xaxis=dict(tickvals=[0, 1], ticktext=['0', '1']),
    showlegend=False
)

fig.show()
```

[9]:
```
train_gender_counts = df_train['Gender'].value_counts(normalize=True)
test_gender_counts = df_test['Gender'].value_counts(normalize=True)



gender_colors = {'Male': '#3d2bff', 'Female': '#ff0070'}

fig = make_subplots(rows=1, cols=2, subplot_titles=("Train Data", "Test Data"))

fig.add_trace(
    go.Bar(
        x=train_gender_counts.index,
        y=train_gender_counts.values,
        marker_color=[gender_colors[gender] for gender in train_gender_counts.
 ↪index],
        name='Train'
    ),
    row=1, col=1
)

fig.add_trace(
    go.Bar(
        x=test_gender_counts.index,
        y=test_gender_counts.values,
        marker_color=[gender_colors[gender] for gender in test_gender_counts.
 ↪index],
        name='Test'
    ),
    row=1, col=2
)
fig.update_yaxes(title_text='Percentage (%)', tickformat='.2%', row=1, col=1)
fig.update_yaxes(title_text='Percentage (%)', tickformat='.2%', row=1, col=2)

fig.update_layout(
    title_text='Normalized Gender Distribution in Train and Test Datasets',
    showlegend=False
)

fig.show()
```

[10]:
```
train_license_counts = df_train['Driving_License'].value_counts(normalize=True)
test_license_counts = df_test['Driving_License'].value_counts(normalize=True)

license_colors = {1: '#004c6d', 0: '#ff0070'}


fig = make_subplots(rows=1, cols=2, subplot_titles=("Train Data", "Test Data"))
```

```
fig.add_trace(
    go.Bar(
        x=train_license_counts.index,
        y=train_license_counts.values,
        marker_color=[license_colors[license] for license in␣
 ↪train_license_counts.index],
        name='Train'
    ),
    row=1, col=1
)

fig.add_trace(
    go.Bar(
        x=test_license_counts.index,
        y=test_license_counts.values,
        marker_color=[license_colors[license] for license in␣
 ↪test_license_counts.index],
        name='Test'
    ),
    row=1, col=2
)

fig.update_yaxes(title_text='Percentage (%)', tickformat='.2%', row=1, col=1)
fig.update_yaxes(title_text='Percentage (%)', tickformat='.2%', row=1, col=2)
fig.update_xaxes(tickvals=[0, 1], ticktext=['0', '1'], row=1, col=1)
fig.update_xaxes(tickvals=[0, 1], ticktext=['0', '1'], row=1, col=2)

fig.update_layout(
    title_text='Normalized Distribution of Driving License in Train and Test␣
 ↪Datasets',
    showlegend=False
)

fig.show()
```

[11]:
```
normalized_counts = df_train.groupby('Driving_License')['Response'].
 ↪value_counts(normalize=True).unstack().fillna(0)


normalized_counts_long = normalized_counts.reset_index().
 ↪melt(id_vars='Driving_License', var_name='Response', value_name='Proportion')

color_map = {0: '#23b2ff', 1: '#f28bb6'}

fig = px.bar(normalized_counts_long, x='Driving_License', y='Proportion',␣
 ↪color='Response',
```

```
            color_discrete_map=color_map,
            labels={'Driving_License': 'Driving License', 'Proportion':␣
 ↪'Proportion'},
            title='Normalized Proportion of Response by Driving License')

fig.update_layout(
    barmode='group',
    xaxis_title='Driving License',
    yaxis_title='Proportion'
)


fig.show()
```

[12]:
```
import pandas as pd

def find_absent_values(train_df, test_df, column_name):

    train_values = set(train_df[column_name].unique())
    test_values = set(test_df[column_name].unique())

    values_in_train_not_in_test = train_values - test_values

    values_in_train_not_in_test_list = list(values_in_train_not_in_test)

    print(f"{column_name} values present in train but absent in test:",␣
 ↪values_in_train_not_in_test_list)



    return values_in_train_not_in_test_list
```

[13]:
```
values_in_train_not_in_test_list_region =␣
 ↪find_absent_values(df_train,df_test,'Region_Code')
```

```
Region_Code values present in train but absent in test: [39.2]
```

[14]:
```
df_train[df_train['Region_Code'].isin(values_in_train_not_in_test_list_region)]
```

[14]:

	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle_Damage	Annual_Premium	Policy_Sales_Channel	Vintage	Response
11370234	Female	20	1	39.2	1	< 1 Year	No	2630.0	159.0	74	0

[15]:
```
df_train.loc[df_train['Region_Code'].
 ↪isin(values_in_train_not_in_test_list_region),'Region_Code'] =39.0
```

[16]:
```
values_in_train_not_in_test_list_policy =␣
 ↪find_absent_values(df_train,df_test,'Policy_Sales_Channel')
```

```
Policy_Sales_Channel values present in train but absent in test: [33.0, 5.0,
6.0]
```

[17]:
```
df_train[df_train['Policy_Sales_Channel'].
 ↪isin(values_in_train_not_in_test_list_policy)]
```

[17]:

	Gender	Age	Driving_License	Region_Code	Previously_Insured	\
202578	Female	62	1	50.0	1	
451535	Female	22	1	8.0	0	
844680	Female	53	1	28.0	0	
2674524	Female	46	1	28.0	0	
3613846	Male	46	1	28.0	0	
9854728	Male	36	1	28.0	0	
10981127	Male	38	1	14.0	1	

	Vehicle_Age	Vehicle_Damage	Annual_Premium	Policy_Sales_Channel	\
202578	1-2 Year	No	36628.0	33.0	
451535	< 1 Year	Yes	39495.0	6.0	
844680	1-2 Year	Yes	44302.0	33.0	
2674524	> 2 Years	Yes	44302.0	33.0	
3613846	1-2 Year	Yes	44302.0	33.0	
9854728	> 2 Years	Yes	30203.0	5.0	
10981127	1-2 Year	No	31458.0	33.0	

	Vintage	Response
202578	97	0
451535	189	0
844680	76	0
2674524	56	0
3613846	76	1
9854728	109	0
10981127	33	0

[18]:
```
df_train = df_train[~df_train['Policy_Sales_Channel'].
 ↪isin(values_in_train_not_in_test_list_policy)].reset_index(drop=True)
```

[19]:
```
find_absent_values(df_train,df_test,'Vintage')
```

```
Vintage values present in train but absent in test: []
```

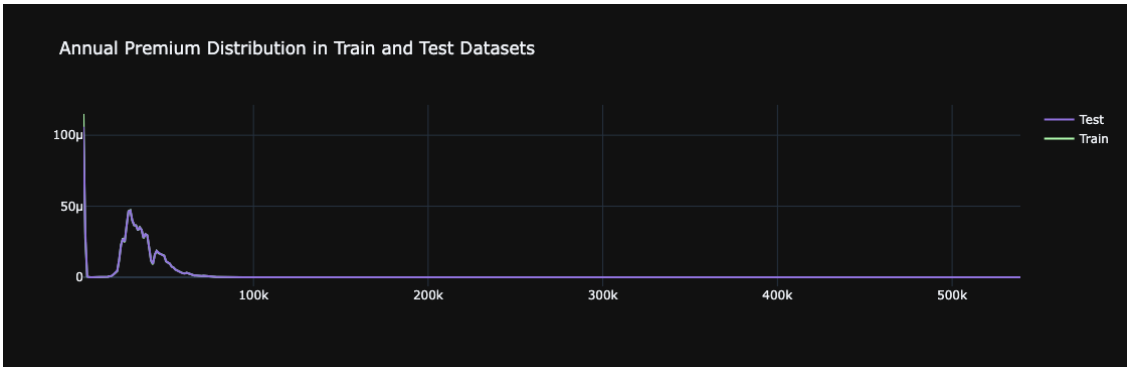
[19]:
```
[]
```

[20]:
```
train_vehicle_age_counts = df_train['Vehicle_Age'].value_counts(normalize=True)
test_vehicle_age_counts = df_test['Vehicle_Age'].value_counts(normalize=True)


vehicle_age_colors = {'1-2 Year': '#1f6271', '> 2 Years': '#6fa6dd', '< 1 Year':
 ↪ '#d1b8ff'}


fig = make_subplots(rows=1, cols=2, subplot_titles=("Train Data", "Test Data"))


fig.add_trace(
    go.Bar(
        x=train_vehicle_age_counts.index,
        y=train_vehicle_age_counts.values,
        marker_color=[vehicle_age_colors[age] for age in␣
 ↪train_vehicle_age_counts.index],
        name='Train'
    ),
    row=1, col=1
)

fig.add_trace(
    go.Bar(
        x=test_vehicle_age_counts.index,
        y=test_vehicle_age_counts.values,
        marker_color=[vehicle_age_colors[age] for age in␣
 ↪test_vehicle_age_counts.index],
        name='Test'
    ),
    row=1, col=2
)

fig.update_yaxes(title_text='Percentage (%)', tickformat='.2%', row=1, col=1)
fig.update_yaxes(title_text='Percentage (%)', tickformat='.2%', row=1, col=2)


fig.update_layout(
    title_text='Normalized Vehicle Age Distribution in Train and Test Datasets',
    showlegend=False
)

fig.show()
```

[21]:
```
name ='Age'
hist_data = [df_train[name],df_test[name]]
```

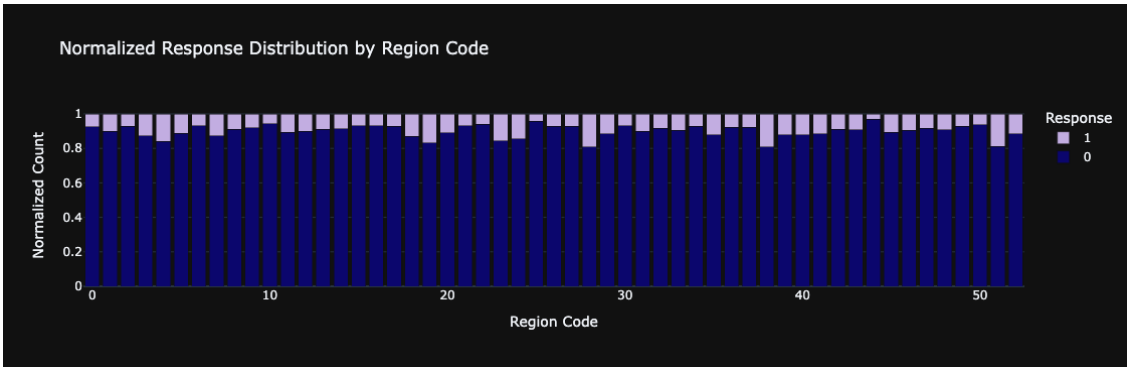
```
group_labels = ['Train','Test']
colors = ['#b3ffb2','#9173de']


fig = ff.create_distplot(hist_data, group_labels, show_hist=False,␣
 ↪colors=colors,show_rug=False)


fig.update_layout(title_text='Age Distribution in Train and Test Datasets')
fig.show()
```

[22]:
```
name ='Annual_Premium'
hist_data = [df_train[name],df_test[name]]


group_labels = ['Train','Test']
colors = ['#b3ffb2','#9173de']


fig = ff.create_distplot(hist_data, group_labels, show_hist=False,␣
 ↪colors=colors,show_rug=False)


fig.update_layout(title_text='Annual Premium Distribution in Train and Test␣
 ↪Datasets')
fig.show()
```



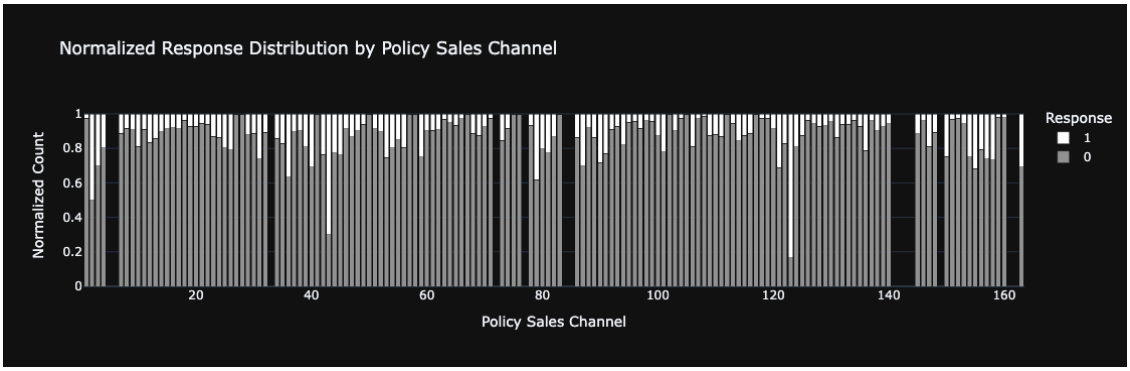
[23]:
```
grouped_data = df_train.groupby(['Region_Code', 'Response'])['Response'].
 ↪count().unstack().fillna(0)
normalized_data = grouped_data.div(grouped_data.sum(axis=1), axis=0).
 ↪reset_index()
```

```
# Melt the DataFrame for Plotly
melted_data = normalized_data.melt(id_vars='Region_Code', value_vars=[0, 1],␣
 ↪var_name='Response', value_name='Normalized_Count')
melted_data['Response'] = melted_data['Response'].astype(int)

# Plot
fig = go.Figure()

# Add traces for each 'Response' value
for response in melted_data['Response'].unique():
    df_response = melted_data[melted_data['Response'] == response]
    fig.add_trace(
        go.Bar(
            x=df_response['Region_Code'],
            y=df_response['Normalized_Count'],
            name=f'{response}',
            marker_color='#0b066d' if response == 0 else '#c2ade1'
        )
    )

# Update layout
fig.update_layout(
    title='Normalized Response Distribution by Region Code',
    xaxis_title='Region Code',
    yaxis_title='Normalized Count',
    barmode='stack',
    legend_title='Response'
)

fig.show()
```

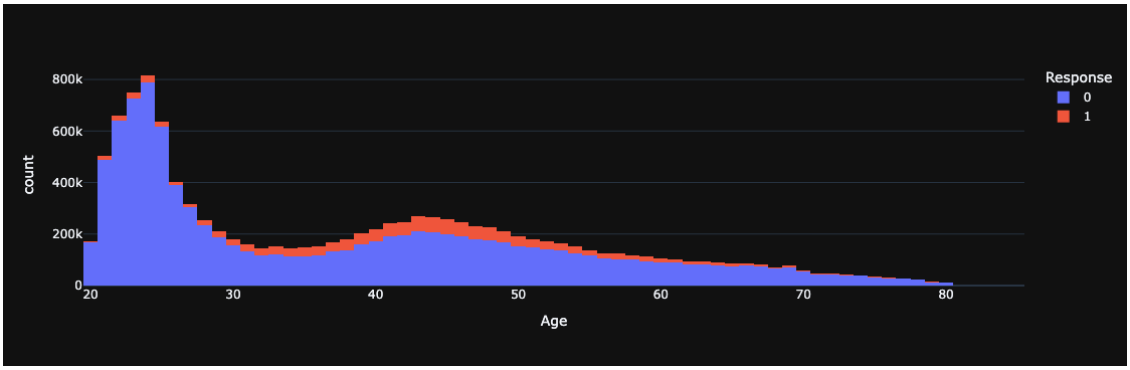


[24]:
```
# Group by 'Policy_Sales_Channel' and 'Response', calculate normalized counts
grouped_data = df_train.groupby(['Policy_Sales_Channel',␣
 ↪'Response'])['Response'].count().unstack().fillna(0)
```

```
normalized_data = grouped_data.div(grouped_data.sum(axis=1), axis=0).
 ↪reset_index()

# Melt the DataFrame for Plotly
melted_data = normalized_data.melt(id_vars='Policy_Sales_Channel',␣
 ↪value_vars=[0, 1], var_name='Response', value_name='Normalized_Count')
melted_data['Response'] = melted_data['Response'].astype(int)

# Plot
fig = go.Figure()

# Add traces for each 'Response' value
for response in melted_data['Response'].unique():
    df_response = melted_data[melted_data['Response'] == response]
    fig.add_trace(
        go.Bar(
            x=df_response['Policy_Sales_Channel'],
            y=df_response['Normalized_Count'],
            name=f'{response}',
            marker_color='#919191' if response == 0 else '#ffffff'
        )
    )

# Update layout
fig.update_layout(
    title='Normalized Response Distribution by Policy Sales Channel',
    xaxis_title='Policy Sales Channel',
    yaxis_title='Normalized Count',
    barmode='stack',
    legend_title='Response'
)

fig.show()
```



[25]:
```
fig = px.histogram(df_train, x="Age", color="Response")
fig.show()
```



[26]:
```
normalized_counts = df_train.groupby('Previously_Insured')['Response'].
 ↪value_counts(normalize=True).unstack().fillna(0)


normalized_counts_long = normalized_counts.reset_index().
 ↪melt(id_vars='Previously_Insured', var_name='Response',␣
 ↪value_name='Proportion')

color_map = {0: '#23b2ff', 1: '#f28bb6'}

fig = px.bar(normalized_counts_long, x='Previously_Insured', y='Proportion',␣
 ↪color='Response',
             color_discrete_map=color_map,
             labels={'Previously_Insured': 'Previously Insured', 'Proportion':␣
 ↪'Proportion'},
             title='Normalized Proportion of Response by Previously Insured')

fig.update_layout(
    barmode='group',
    xaxis_title='Previously Insured',
    yaxis_title='Proportion'
)


fig.show()
```

[27]:
```
def sanitize_column_names(columns):

    sanitized_columns = [re.sub(r'[^\w\s]', '', col) for col in columns]
    return sanitized_columns
```

[28]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.compose import make_column_selector as selector
import re
def preprocess_data(train, test, target_column):

    X = train.drop(target_column, axis=1)
    y = train[target_column]

    X_train, X_test, y_train, y_test = train_test_split(X, y,stratify=y,␣
 ↪test_size=0.2, random_state=42)

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(), selector(dtype_exclude='object')),
            ('cat', OneHotEncoder(drop='first', sparse_output=False),␣
 ↪selector(dtype_include='object'))
        ]
    )


    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor)
    ])

    X_train = pipeline.fit_transform(X_train)
    X_test = pipeline.transform(X_test)
    X_test_final = pipeline.transform(test)

    num_cols = X.select_dtypes(exclude='object').columns
    cat_cols = pipeline.named_steps['preprocessor'].transformers_[1][1].
 ↪get_feature_names_out(X.select_dtypes(include='object').columns)

    all_cols = list(num_cols) + list(cat_cols)

    sanitized_cols = sanitize_column_names(all_cols)

    X_train_df = pd.DataFrame(X_train, columns=sanitized_cols)
    X_test_df = pd.DataFrame(X_test, columns=sanitized_cols)
    X_test_final_df = pd.DataFrame(X_test_final, columns=sanitized_cols)

    return X_train_df, X_test_df, y_train, y_test, X_test_final_df
```

[29]:
```
X_train_df, X_test_df, y_train, y_test, X_test_final_df =␣
↪preprocess_data(df_train,df_test,'Response')
```

[30]:
```
def convert_columns_to_int(df, columns):

    for column in columns:
        try:

            df[column] = pd.to_numeric(df[column], errors='raise').astype(int)
            print(f"Column '{column}' has been successfully converted to int.")
        except ValueError as e:
            print(f"Error: Unable to convert column '{column}' to int. {e}")
        except KeyError:
            print(f"Error: Column '{column}' not found in DataFrame.")

    return df
```

[31]:
```
col =['Gender_Male','Vehicle_Age_ 1 Year','Vehicle_Age_ 2␣
↪Years','Vehicle_Damage_Yes']
X_train_df = convert_columns_to_int(X_train_df, col)
X_test_df =convert_columns_to_int(X_test_df, col)
X_test_final_df = convert_columns_to_int(X_test_final_df, col)
```

```
Column 'Gender_Male' has been successfully converted to int.
Column 'Vehicle_Age_ 1 Year' has been successfully converted to int.
Column 'Vehicle_Age_ 2 Years' has been successfully converted to int.
Column 'Vehicle_Damage_Yes' has been successfully converted to int.
Column 'Gender_Male' has been successfully converted to int.
Column 'Vehicle_Age_ 1 Year' has been successfully converted to int.
Column 'Vehicle_Age_ 2 Years' has been successfully converted to int.
Column 'Vehicle_Damage_Yes' has been successfully converted to int.
Column 'Gender_Male' has been successfully converted to int.
Column 'Vehicle_Age_ 1 Year' has been successfully converted to int.
Column 'Vehicle_Age_ 2 Years' has been successfully converted to int.
Column 'Vehicle_Damage_Yes' has been successfully converted to int.
```

[32]:
```
def optimize_float_int_memory(df):

    start_mem = df.memory_usage().sum() / 1024**2
    print(f'Memory usage of dataframe is {start_mem:.2f} MB')

    for col in df.columns:
        col_type = df[col].dtype

        if col_type == 'int64':
            c_min = df[col].min()
            c_max = df[col].max()
            if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
```

```
                df[col] = df[col].astype(np.int8)
            elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).
↪max:
                df[col] = df[col].astype(np.int16)
            elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).
↪max:
                df[col] = df[col].astype(np.int32)

        elif col_type == 'float64':
            c_min = df[col].min()
            c_max = df[col].max()
            if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.
↪float16).max:
                df[col] = df[col].astype(np.float16)
            else:
                df[col] = df[col].astype(np.float32)

    end_mem = df.memory_usage().sum() / 1024**2
    print(f'Memory usage after optimization is: {end_mem:.2f} MB')
    print(f'Decreased by {(100 * (start_mem - end_mem) / start_mem):.1f}%')

    return df
```

[33]:
```
X_train_df = optimize_float_int_memory(X_train_df)
X_test_df = optimize_float_int_memory(X_test_df)
X_test_final_df = optimize_float_int_memory(X_test_final_df)
```

```
Memory usage of dataframe is 772.42 MB
Memory usage after optimization is: 157.99 MB
Decreased by 79.5%
Memory usage of dataframe is 193.10 MB
Memory usage after optimization is: 39.50 MB
Decreased by 79.5%
Memory usage of dataframe is 643.68 MB
Memory usage after optimization is: 131.66 MB
Decreased by 79.5%
```

[35]:
```
import umap
```

[ ]:
```
umap_reducer = umap.UMAP(n_jobs=-1).fit(X_train_df)

X_umap = umap_reducer.transform(X_train_df)
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use
omp_set_max_active_levels instead.
```

[ ]:
```
plt.figure(figsize=(12, 6))
plt.scatter(X_umap[:, 0], X_umap[:, 1], c=y_train, s=10)
```

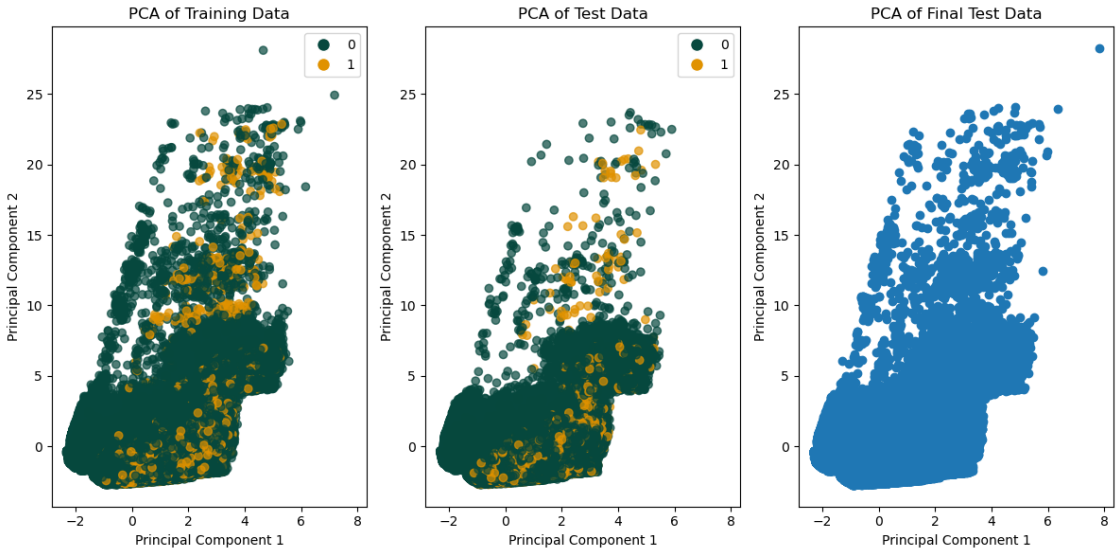
```
plt.title('UMAP Projection')
plt.xlabel('UMAP1')
plt.ylabel('UMAP2')
plt.show()
```

[32]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)

X_train_pca = pca.fit_transform(X_train_df)
X_test_pca = pca.transform(X_test_df)

df_train_pca = pd.DataFrame(X_train_pca, columns=['PC1', 'PC2'])


df_test_pca = pd.DataFrame(X_test_pca, columns=['PC1', 'PC2'])
df_test_pca_final = pd.DataFrame(pca.transform(X_test_final_df),␣
 ↪columns=['PC1', 'PC2'])
```

[33]:
```
df_train_pca['Response']= y_train.reset_index(drop=True)
df_test_pca['Response']=  y_test.reset_index(drop=True)
```

[34]:
```
colors = {0: '#06483e', 1: '#e09200'}
opacity = 0.7

plt.figure(figsize=(12, 6))

ax1 = plt.subplot(1, 3, 1)
scatter_train = ax1.scatter(df_train_pca['PC1'], df_train_pca['PC2'],
                            c=df_train_pca['Response'].map(colors),␣
 ↪alpha=opacity)
ax1.set_title('PCA of Training Data')
ax1.set_xlabel('Principal Component 1')
ax1.set_ylabel('Principal Component 2')

ax2 = plt.subplot(1, 3, 2, sharex=ax1, sharey=ax1)
scatter_test = ax2.scatter(df_test_pca['PC1'], df_test_pca['PC2'],
                           c=df_test_pca['Response'].map(colors), alpha=opacity)
ax2.set_title('PCA of Test Data')
ax2.set_xlabel('Principal Component 1')
ax2.set_ylabel('Principal Component 2')

ax3 = plt.subplot(1, 3, 3, sharex=ax2, sharey=ax2)
scatter_test_final = ax3.scatter(df_test_pca_final['PC1'],␣
 ↪df_test_pca_final['PC2']
                            )
ax3.set_title('PCA of Final Test Data')
ax3.set_xlabel('Principal Component 1')
```

```
ax3.set_ylabel('Principal Component 2')

handles = [plt.Line2D([0], [0], marker='o', color='w',␣
 ↪markerfacecolor='#06483e', markersize=10, label='0'),
           plt.Line2D([0], [0], marker='o', color='w',␣
 ↪markerfacecolor='#e09200', markersize=10, label='1')]

ax1.legend(handles=handles, loc='upper right')
ax2.legend(handles=handles, loc='upper right')

plt.tight_layout()
plt.show()
```



[35]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.compose import make_column_selector as selector
from sklearn.metrics import roc_auc_score
from sklearn.base import ClassifierMixin
from sklearn.utils import all_estimators


# Importing external classifiers
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
from xgboost import XGBClassifier
```

```
# Import classifiers from sklearn manually
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier,␣
↪GradientBoostingClassifier, AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
```

[37]:
```
def train_and_evaluate_classifiers(X_train, y_train, X_test, y_test):

    results = []


    classifiers = [ ('Logistic Regression',␣
↪LogisticRegression(max_iter=1000,random_state=42)),
        ('Decision Tree', DecisionTreeClassifier(random_state=42)),
        ('LightGBM', LGBMClassifier(verbose=-1,random_state=42)),
        ('CatBoost', CatBoostClassifier(verbose=0,random_state=42)),
        ('XGBoost', XGBClassifier(use_label_encoder=False,␣
↪eval_metric='logloss',random_state=42))
    ]


    for name, clf in classifiers:

        try:
            print(f"Training {name}...")
            clf.fit(X_train, y_train)

            y_train_pred_proba = clf.predict_proba(X_train)[:, 1]
            y_test_pred_proba = clf.predict_proba(X_test)[:, 1]

            train_auc = roc_auc_score(y_train, y_train_pred_proba)
            test_auc = roc_auc_score(y_test, y_test_pred_proba)

            results.append({'Classifier': name, 'Train AUC': train_auc, 'Test␣
↪AUC': test_auc})
        except Exception as e:
            print(f"Classifier {name} failed: {str(e)}")
```

```
            continue

    results_df = pd.DataFrame(results).sort_values(by='Test AUC',␣
 ↪ascending=False).reset_index(drop=True)

    return results_df
```

[38]:
```
results_df = train_and_evaluate_classifiers(X_train_df, y_train, X_test_df,␣
 ↪y_test)
```

```
Training Logistic Regression…
Training Decision Tree…
Training LightGBM…
Training CatBoost…
Training XGBoost…
```

[39]:
```
results_df
```

[39]:

	Classifier	Train AUC	Test AUC
0	CatBoost	0.882829	0.880795
1	XGBoost	0.878968	0.878407
2	LightGBM	0.875757	0.875885
3	Logistic Regression	0.841208	0.841739
4	Decision Tree	1.000000	0.622992

[40]:
```
def evaluate_catboost(X_train, y_train, X_test, y_test, X_test_final):

    model = CatBoostClassifier(verbose=0,iterations=200)


    print("Training CatBoostClassifier...")
    model.fit(X_train, y_train)


    y_train_pred_proba = model.predict_proba(X_train)[:, 1]
    y_test_pred_proba = model.predict_proba(X_test)[:, 1]

    y_test_final_pred_proba = model.predict_proba(X_test_final)[:, 1]

    train_auc = roc_auc_score(y_train, y_train_pred_proba)
    test_auc = roc_auc_score(y_test, y_test_pred_proba)


    print(f"Train AUC: {train_auc:.4f}")
    print(f"Test AUC: {test_auc:.4f}")

    return y_test_final_pred_proba
```

```
y_test_final_pred_proba = evaluate_catboost(X_train_df, y_train, X_test_df,␣
↪y_test, X_test_final_df)
```

```
Training CatBoostClassifier…
Train AUC: 0.8790
Test AUC: 0.8789
```

[46]:
```
def optimize_catboost(xtrain, xtest, ytrain, ytest, n_trials=5, seed=42):



    def objective(trial):

        params = {
            'iterations': trial.suggest_int('iterations', 100, 1000),
            'depth': trial.suggest_int('depth', 3, 10),
            'learning_rate': trial.suggest_loguniform('learning_rate', 1e-4,␣
↪1e-1),
            'l2_leaf_reg': trial.suggest_loguniform('l2_leaf_reg', 1e-5, 1e2),
            'border_count': trial.suggest_int('border_count', 32, 255),
            'random_strength': trial.suggest_loguniform('random_strength',␣
↪1e-5, 10),
            'bagging_temperature': trial.
↪suggest_loguniform('bagging_temperature', 0.01, 10),


        }

        model = CatBoostClassifier(**params,random_seed=seed)
        model.fit(xtrain, ytrain, eval_set=(xtest, ytest),␣
↪early_stopping_rounds=50, verbose=0)

        preds = model.predict_proba(xtest)[:, 1]
        auc = roc_auc_score(ytest, preds)

        return auc

    study = optuna.create_study(direction="maximize", sampler=optuna.samplers.
↪TPESampler(seed=seed))
    study.optimize(objective, n_trials=n_trials)

    print(f"Best AUC: {study.best_value}")
    print(f"Best hyperparameters: {study.best_params}")

    return study.best_params
```

[47]:
```
best_params = optimize_catboost(X_train_df, X_test_df, y_train, y_test,␣
↪n_trials=5, seed=42)
```

```
Best AUC: 0.8676188341814324
Best hyperparameters: {'iterations': 437, 'depth': 10, 'learning_rate':
0.015702970884055395, 'l2_leaf_reg': 0.155099139875943, 'border_count': 66,
'random_strength': 8.629132190071849e-05, 'bagging_temperature':
0.014936568554617643}
```

[50]:
```
def evaluate_catboost(X_train, y_train, X_test, y_test, X_test_final,␣
↪best_params):
    model = CatBoostClassifier(verbose=0, **best_params)

    print("Training CatBoostClassifier with best hyperparameters...")
    model.fit(X_train, y_train)

    y_train_pred_proba = model.predict_proba(X_train)[:, 1]
    y_test_pred_proba = model.predict_proba(X_test)[:, 1]
    y_test_final_pred_proba = model.predict_proba(X_test_final)[:, 1]

    train_auc = roc_auc_score(y_train, y_train_pred_proba)
    test_auc = roc_auc_score(y_test, y_test_pred_proba)

    print(f"Train AUC: {train_auc:.4f}")
    print(f"Test AUC: {test_auc:.4f}")

    return y_test_final_pred_proba
```

[52]:
```
y_test_final_pred_proba = evaluate_catboost(X_train_df, y_train, X_test_df,␣
↪y_test, X_test_final_df,best_params)
```

```
Training CatBoostClassifier with best hyperparameters…
Train AUC: 0.8671
Test AUC: 0.8674
```

[53]:
```
def save_predictions_to_csv(y_test_prob,␣
↪output_csv_name,input_csv='sample_submission.csv'):
    submission_df = pd.read_csv(input_csv)

    submission_df['Response'] = y_test_prob

    submission_df.to_csv(output_csv_name, index=False)
    print(f"Predictions saved to {output_csv_name}")
```

[ ]:
```

```

[54]:
```
save_predictions_to_csv(y_test_final_pred_proba,'17thsept.csv')
```

```
Predictions saved to 17thsept.csv
```

[ ]: