Artificial Intelligence - MSc

ED5005 - Project and Dissertation SEM 2023_4

Title: "The AI Augmented Underwriter - Integrating unsupervised learning for an understandable approach to predictive insurance underwriting."

- "Integrating unsupervised learning techniques for an interpretable, explainable approach to predictive insurance underwriting."
- "Integrating unsupervised and supervised learning with human-in-the-loop to increase interpretability and explainability in predictive underwriting"
- "Integrating unsupervised and supervised learning with human-in-the-loop as an explainable approach to augmented underwriting in Life Insurance"
- "Integrating Unsupervised Learning and Human-in-the-Loop for Explainable Predictive Modelling in Life Insurance Underwriting"

Supervisor: Martin Cunneen

Student Name: Paul Trayers

Student ID: 8907021

Date of Final Submission: Aug 4th 2024

Introduction

Double-click (or enter) to edit

For this project...

We load this dataset directly from a URL. AutoGluon's `TabularDataset` is a subclass of pandas [Prudential Life Insurance](), so any `DataFrame` methods can be used on `TabularDataset` as well.

1. Exploratory Data Analysis (EDA)

1.1 Import Libraries

```
# Import key modules that will be used throughout the project.

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # graphs/plotting
import seaborn as sns
```

1.2 Import Dataset

```
import pandas as pd
from google.colab import drive

drive.mount('/content/drive/', force_remount=True)

# Load the CSV data file into a Pandas dataframe.
df = pd.read_csv('/content/drive/My Drive/ED5005_Project/data/train.csv')

# Review main characteristics of the Pandas dataframe.
print(df.dtypes.to_string())
df.describe()
```

```
Mounted at /content/drive/
Id                          int64
Product_Info_1              int64
Product_Info_2             object
Product_Info_3              int64
Product_Info_4            float64
Product_Info_5              int64
Product_Info_6              int64
Product_Info_7              int64
Ins_Age                   float64
Ht                        float64
Wt                        float64
BMI                       float64
Employment_Info_1         float64
Employment_Info_2           int64
Employment_Info_3           int64
Employment_Info_4         float64
Employment_Info_5           int64
Employment_Info_6         float64
InsuredInfo_1               int64
InsuredInfo_2               int64
InsuredInfo_3               int64
InsuredInfo_4               int64
InsuredInfo_5               int64
InsuredInfo_6               int64
InsuredInfo_7               int64
Insurance_History_1         int64
Insurance_History_2         int64
Insurance_History_3         int64
Insurance_History_4         int64
Insurance_History_5       float64
Insurance_History_7         int64
Insurance_History_8         int64
Insurance_History_9         int64
Family_Hist_1               int64
Family_Hist_2             float64
Family_Hist_3             float64
Family_Hist_4             float64
Family_Hist_5             float64
Medical_History_1         float64
Medical_History_2           int64
Medical_History_3           int64
Medical_History_4           int64
Medical_History_5           int64
Medical_History_6           int64
Medical_History_7           int64
Medical_History_8           int64
Medical_History_9           int64
Medical_History_10        float64
Medical_History_11          int64
Medical_History_12          int64
Medical_History_13          int64
Medical_History_14          int64
Medical_History_15        float64
Medical_History_16          int64
Medical_History_17          int64
Medical_History_18          int64
Medical_History_19          int64
Medical_History_20          int64
Medical_History_21          int64
```

```
Medical_History_22       int64
Medical_History_23       int64
Medical_History_24     float64
Medical_History_25       int64
Medical_History_26       int64
Medical_History_27       int64
Medical_History_28       int64
Medical_History_29       int64
Medical_History_30       int64
Medical_History_31       int64
Medical_History_32     float64
Medical_History_33       int64
Medical_History_34       int64
Medical_History_35       int64
Medical_History_36       int64
Medical_History_37       int64
Medical_History_38       int64
Medical_History_39       int64
Medical_History_40       int64
Medical_History_41       int64
Medical_Keyword_1        int64
Medical_Keyword_2        int64
Medical_Keyword_3        int64
Medical_Keyword_4        int64
Medical_Keyword_5        int64
Medical_Keyword_6        int64
Medical_Keyword_7        int64
Medical_Keyword_8        int64
Medical_Keyword_9        int64
Medical_Keyword_10       int64
Medical_Keyword_11       int64
Medical_Keyword_12       int64
Medical_Keyword_13       int64
Medical_Keyword_14       int64
Medical_Keyword_15       int64
Medical_Keyword_16       int64
Medical_Keyword_17       int64
Medical_Keyword_18       int64
Medical_Keyword_19       int64
Medical_Keyword_20       int64
Medical_Keyword_21       int64
Medical_Keyword_22       int64
Medical_Keyword_23       int64
Medical_Keyword_24       int64
Medical_Keyword_25       int64
Medical_Keyword_26       int64
Medical_Keyword_27       int64
Medical_Keyword_28       int64
Medical_Keyword_29       int64
Medical_Keyword_30       int64
Medical_Keyword_31       int64
Medical_Keyword_32       int64
Medical_Keyword_33       int64
Medical_Keyword_34       int64
Medical_Keyword_35       int64
Medical_Keyword_36       int64
Medical_Keyword_37       int64
Medical_Keyword_38       int64
Medical_Keyword_39       int64
Medical_Keyword_40       int64
Medical_Keyword_41       int64
```

```
Medical_Keyword_41      int64
Medical_Keyword_42      int64
Medical_Keyword_43      int64
Medical_Keyword_44      int64
Medical_Keyword_45      int64
Medical_Keyword_46      int64
Medical_Keyword_47      int64
Medical_Keyword_48      int64
Response                int64
```

	Id	Product_Info_1	Product_Info_3	Product_Info_4	Product_Inf
count	59381.000000	59381.000000	59381.000000	59381.000000	59381.000
mean	39507.211515	1.026355	24.415655	0.328952	2.006
std	22815.883089	0.160191	5.072885	0.282562	0.083
min	2.000000	1.000000	1.000000	0.000000	2.000
25%	19780.000000	1.000000	26.000000	0.076923	2.000
50%	39487.000000	1.000000	26.000000	0.230769	2.000
75%	59211.000000	1.000000	26.000000	0.487179	2.000
max	79146.000000	2.000000	38.000000	1.000000	3.000

8 rows × 127 columns

From the cell above, the dataframe contains different datatypes. Any with `object` dtype contain non-numeric (character) data and must be pre-processed to make machine-interpretable.

```
# set the 'Id' column as the index of the DataFrame
df_index_set = df.set_index('Id')
```

```
df_index_set.head()
```

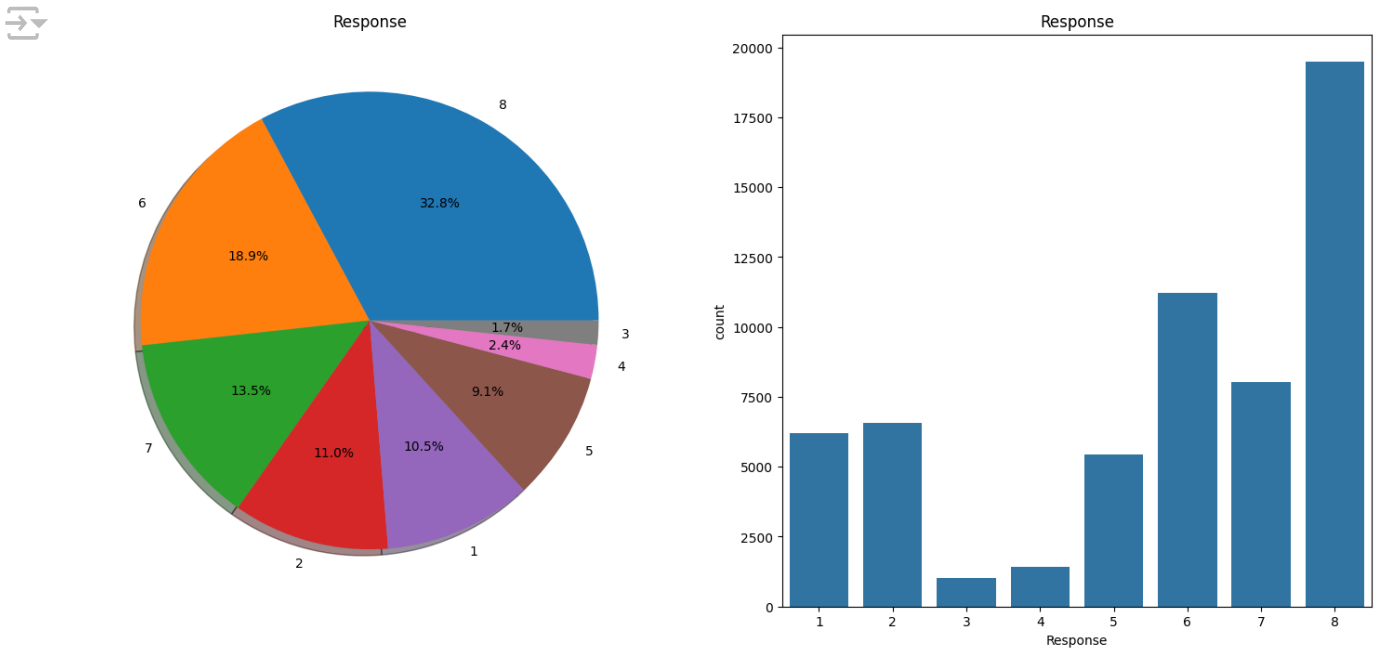
Id	Product_Info_1	Product_Info_2	Product_Info_3	Product_Info_4	Product_Inf
2	1	D3	10	0.076923	
5	1	A1	26	0.076923	
6	1	E1	26	0.076923	
7	1	D4	10	0.487179	
8	1	D2	26	0.230769	

5 rows × 127 columns

The values of each column (risk feature) for the first five life insurance applicants in the dataset are listed in each row, with the Id serving as the index in this case.

As can be observed above, there is a combination of normalized and non-normalized numeric-valued features. Before we train and test our models further, we must examine and, if necessary, pre-process each of these columns.

```
df, ax = plt.subplots(1, 2, figsize=(18, 8))

# Pie chart
df_index_set['Response'].value_counts().plot.pie(autopct='%1.1f%%', ax=ax[0], shac
ax[0].set_title('Response')
ax[0].set_ylabel('')

# Bar chart
sns.countplot(x='Response', data=df_index_set, ax=ax[1])
ax[1].set_title('Response')

plt.show()
```

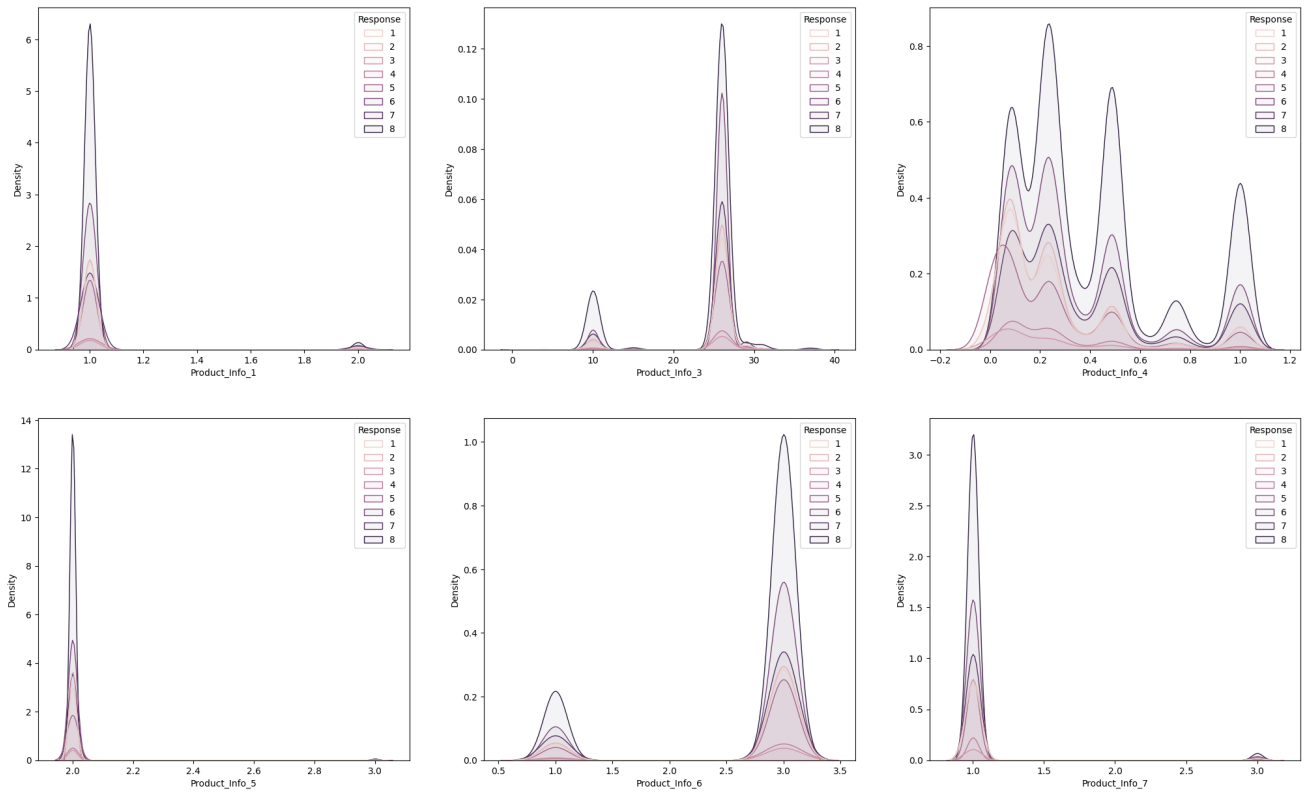


The histogram displays the distribution of applicants across the entire dataset based on their risk ratings (Response).

The distribution is "unbalanced" being skewed towards classes 6–8, which is significant, even though classes 1-2 also make up a sizeable portion of the dataset. Later, we will assess how well our ML models are able to "mimic" this distribution, as an indicator of whether they have been adequately fitted.

1.3 Distributions

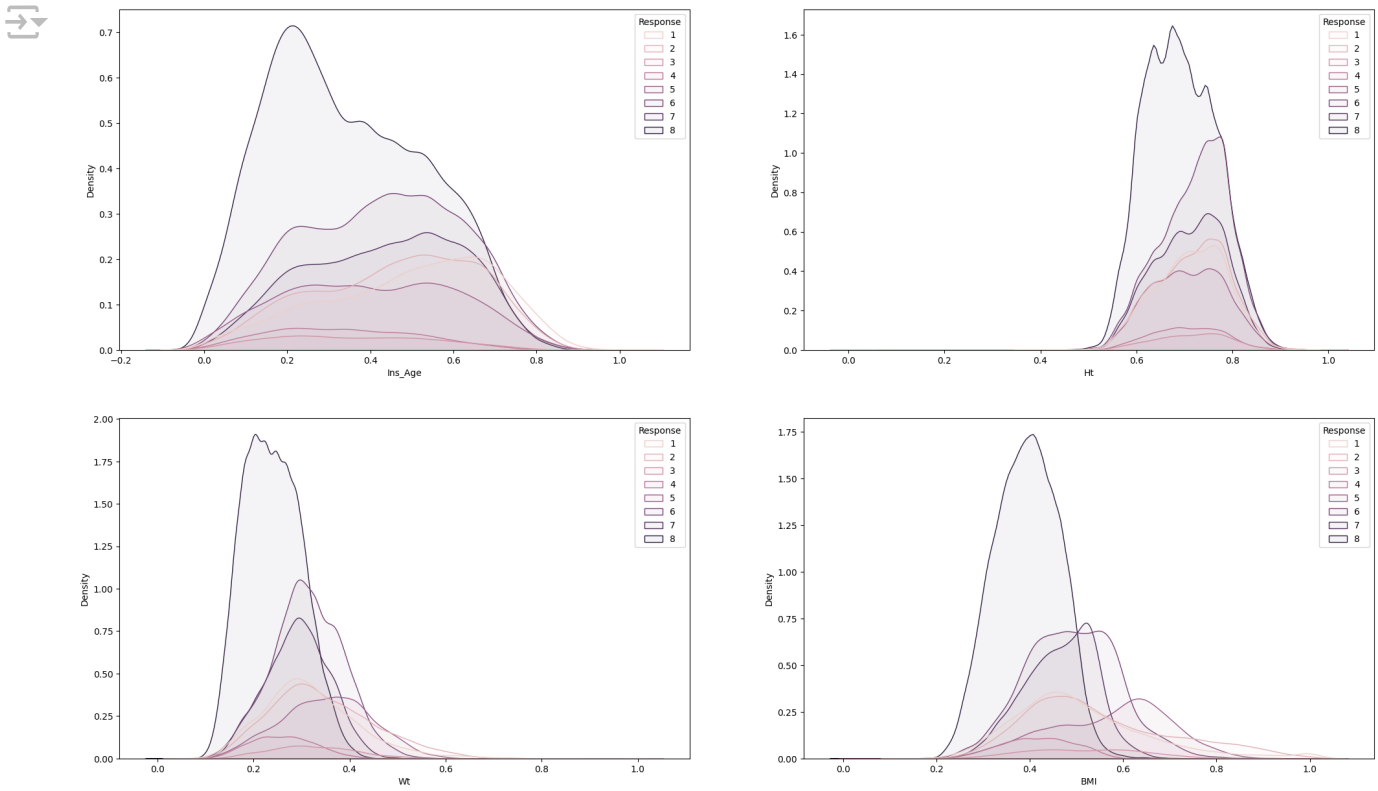
```
# Split out the full set of the main dataset's columns into separate lists for eas

ColSet1_ProdInfo = ['Product_Info_1','Product_Info_2','Product_Info_3','Product_In
ColSet2_ApplicantInfo = ['Ins_Age','Ht','Wt','BMI']
ColSet3_EmploymentInfo = ['Employment_Info_1','Employment_Info_2','Employment_Info
ColSet4_InsuredInfo = ['InsuredInfo_1','InsuredInfo_2','InsuredInfo_3','InsuredInf
ColSet5_InsuranceHistoryInfo = ['Insurance_History_1','Insurance_History_2','Insur
ColSet6_FamilyHistoryInfo = ['Family_Hist_1','Family_Hist_2','Family_Hist_3','Fami

ColSet7_MedicalHistoryInfo = ['Medical_History_1','Medical_History_2','Medical_His
                              'Medical_History_9','Medical_History_10','Medical_Hi
                              'Medical_History_16','Medical_History_17','Medical_H
                              'Medical_History_23','Medical_History_24','Medical_H
                              'Medical_History_30','Medical_History_31','Medical_H
                              'Medical_History_37','Medical_History_38','Medical_H

ColSet8_MedicalKeywordInfo = ['Medical_Keyword_1','Medical_Keyword_2','Medical_Key
                              'Medical_Keyword_9','Medical_Keyword_10','Medical_Ke
                              'Medical_Keyword_17','Medical_Keyword_18','Medical_K
                              'Medical_Keyword_25','Medical_Keyword_26','Medical_K
                              'Medical_Keyword_33','Medical_Keyword_34','Medical_K
                              'Medical_Keyword_41','Medical_Keyword_42','Medical_K


# Set up a subplot grid.
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(25,15))

# Product_Info_2 has been excluded - as this has not yet been encoded into numeric
ColSet1_ProdInfo_kde = ['Product_Info_1','Product_Info_3','Product_Info_4','Produc

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet1_ProdInfo_kde].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])
```



Product Information

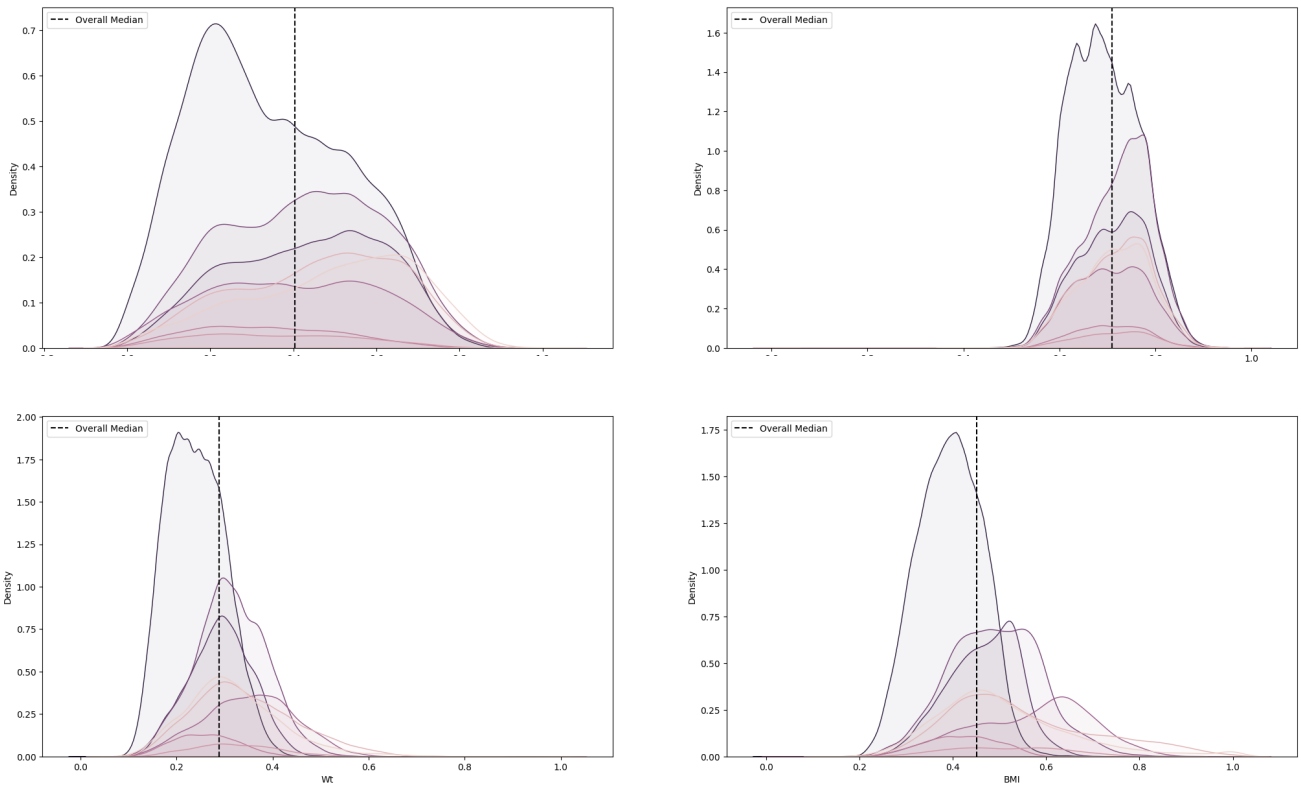
The KDE plots show distributions of Response group/cohort of applicants across various `Product_Info` variables which we can observe have varying modalities. However, they all appear to have a high degree of overlap across the `Response` variable with no major difference in relative densities. Therefore the variations existing between the features do nor particularly help in predicting applicant risk rating.

```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(25,15))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet2_ApplicantInfo].columns):
    plot = sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//2,i%2])
```




```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(25,15))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet2_ApplicantInfo].columns):
    # Create the KDE plot
    plot = sns.kdeplot(data=df_index_set,
                       x=column,
                       hue="Response", fill=True, common_norm=True, alpha=0.05,
```

```
                    ax=axes[i//2,i%2])

    # Calculate the overall median for the column
    overall_median = df_index_set[column].median()

    # Add a vertical line for the overall median
    plot.axvline(overall_median, color='black', linestyle='--', label=f"Overall Me

    # Add legend for the medians AND the response categories
    plot.legend(loc='upper left')
```



Biophysicial Information

Ins_Age: This KDE plot highlights significant variation in the distribution across different Response groups, with each group showing distinct patterns in terms of peak broadening and shouldering. While the majority of the density is spread between x=0 and x=0.9, the shapes and skews of the distributions differ notably among cohorts. For instance, Response category 8

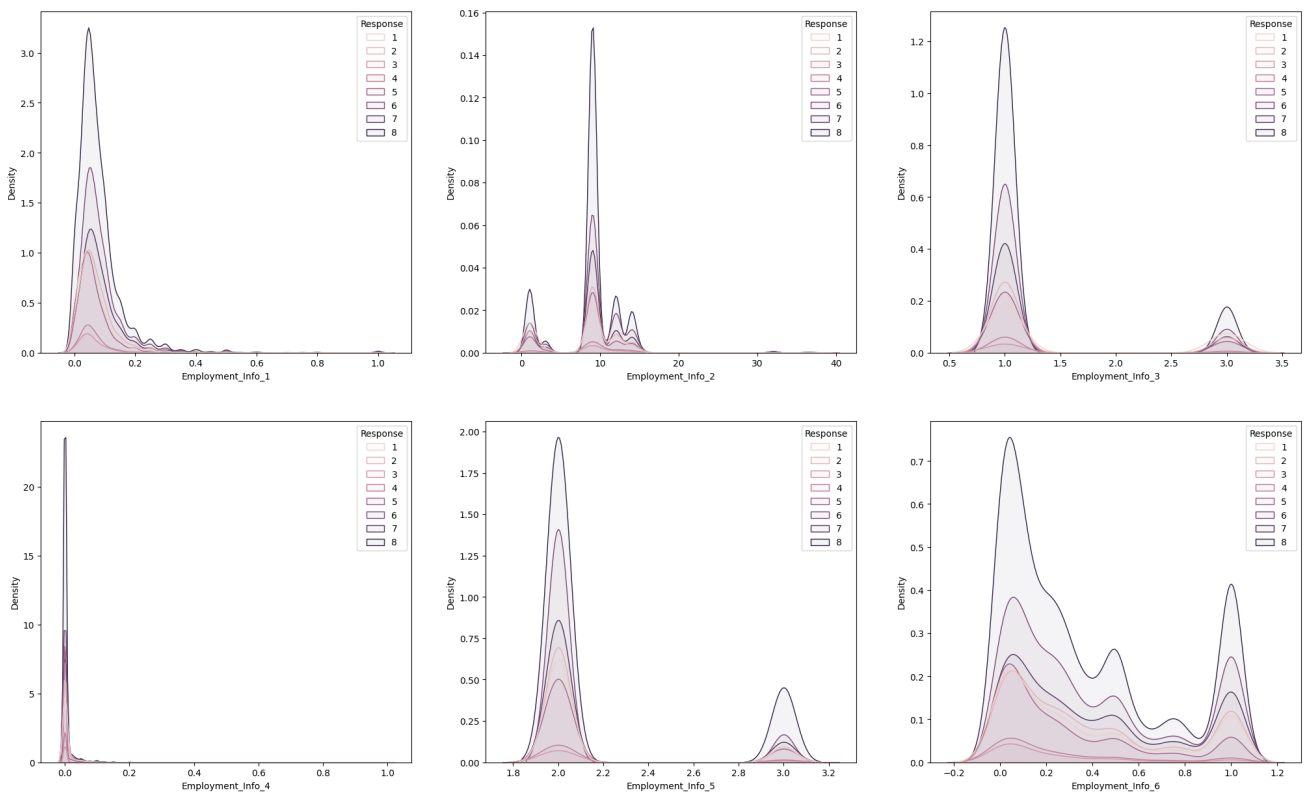
shows a positive skew towards x=0.2, whereas most other categories exhibit negative skews towards x=0.4. Lower Response categories (1 to 4) tend to have broader, more uniform distributions, while higher categories (5 to 8) feature sharper peaks and more pronounced skews. This indicates that as the Response category increases, the age distribution becomes more concentrated around specific ranges. The main demographic insight is that younger individuals (closer to x=0.2) are more prevalent in higher Response categories, whereas a wider age range is seen in lower Response categories.

Ht: This KDE plot highlights significant variation in the 'Ht' distribution across different Response groups, with each group showing distinct patterns in terms of peak broadening and shouldering. The majority of the density is spread between x=0.6 and x=0.8, indicating that most individuals fall within this height range. Each cohort's skew mirrors that seen in the 'Ins_Age' distribution: Response category 8 displays a positive skew, while most other categories exhibit a negative skew. This pattern underscores the demographic insight that individuals in higher Response categories tend to have characteristics concentrated around certain ranges, while lower categories display a broader distribution.

Wt: This KDE plot illustrates significant variation in the `Wt` distribution across different Response groups, with distinct patterns of peak broadening and shouldering. Most of the distribution density lies between x=0.2 and x=0.5. Notably, the distribution for Response category 8 is concentrated around lower 'Wt' values (x=0.2), whereas the rest of the population is more broadly distributed across the 0.2 to 0.5 interval.

BMI: This KDE plot of BMI reveals significant variation in the composition and structure of each Response group distribution, with noticeable broadening and shouldering of the peaks. Similar to the 'Wt' distribution, the majority of the density is spread between x=0.4 and x=0.6. Response category 8 has a distinct peak centered at x=0.4, while most other categories are spread across the x=0.4 to x=0.6 range. Notably, one of the "medium" risk-rating distributions exhibits a pronounced negative skew towards x=0.6, differing from its peers that skew more towards x=0.5.

```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(25,15))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet3_EmploymentInfo].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])
```

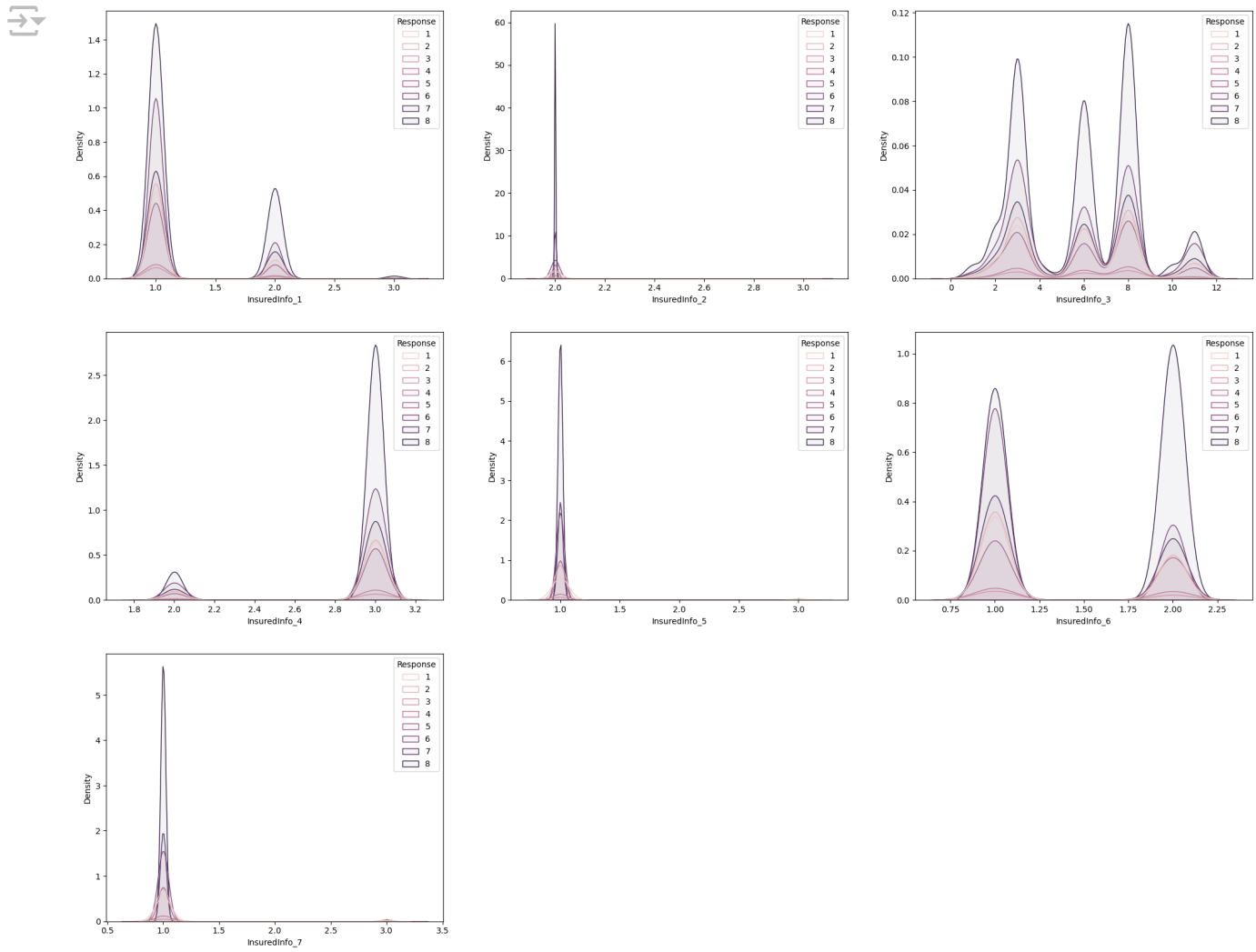


Employment Information

These KDE plots display distributions of `Employment_Info` features with varying modalities, but a common trend is that they closely overlap between each Response group (cohort of applicants), showing no significant differences in relative densities. Therefore, variations in these features are unlikely to contribute individually to predicting an applicant's risk rating.

```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(25,20))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet4_InsuredInfo].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])
```

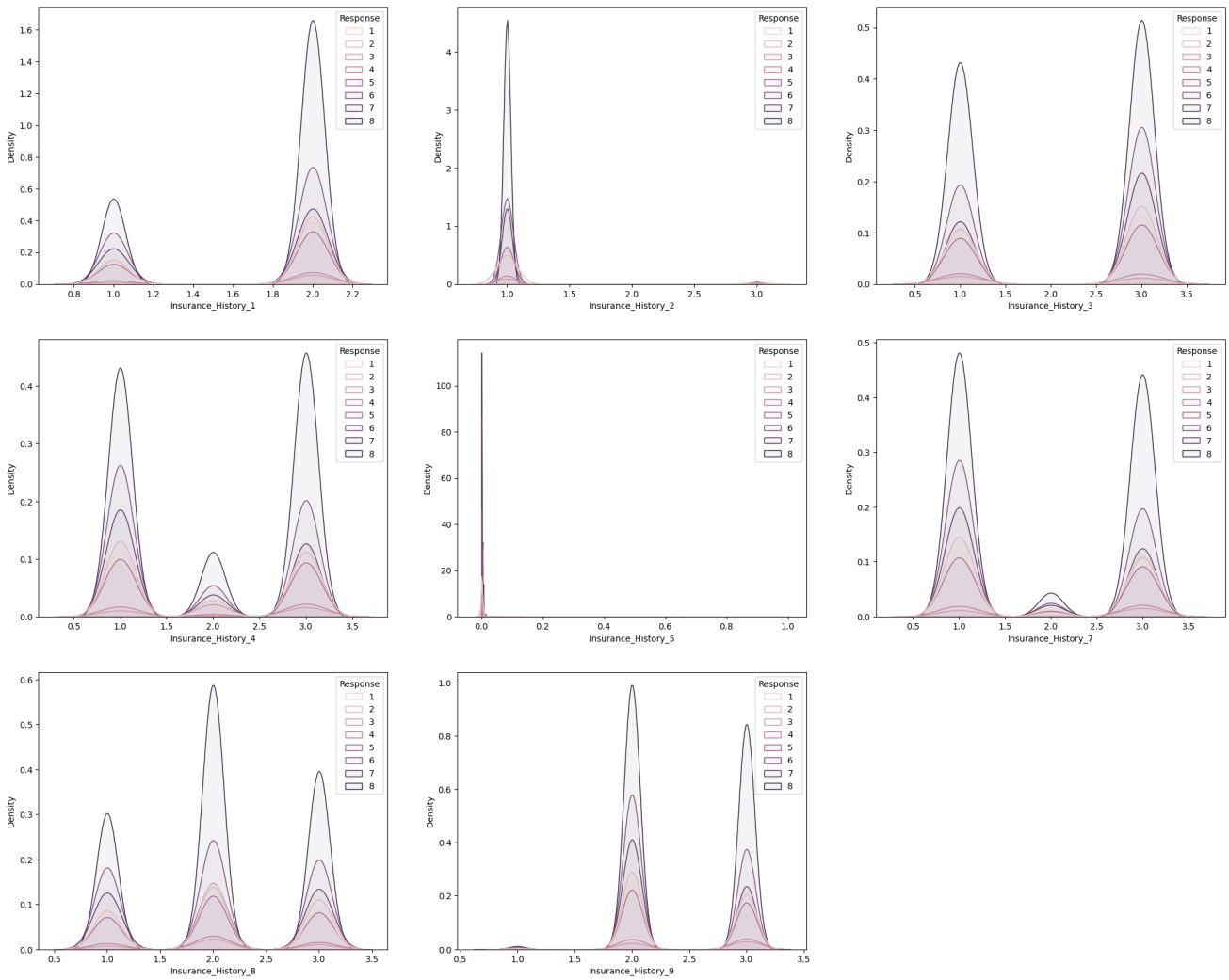
```
# Delete any unused sets of axes in the subplot grid.
fig.delaxes(axes[2,1])
fig.delaxes(axes[2,2])
```



Insured Information

These KDE plots illustrate distributions of various `InsuredInfo` features with varying modalities; however, a consistent trend is that they all closely overlap across different Response groups/cohorts of applicants, with no significant differences in relative densities. Consequently, variations in these features are unlikely to individually aid in predicting an applicant's risk rating.

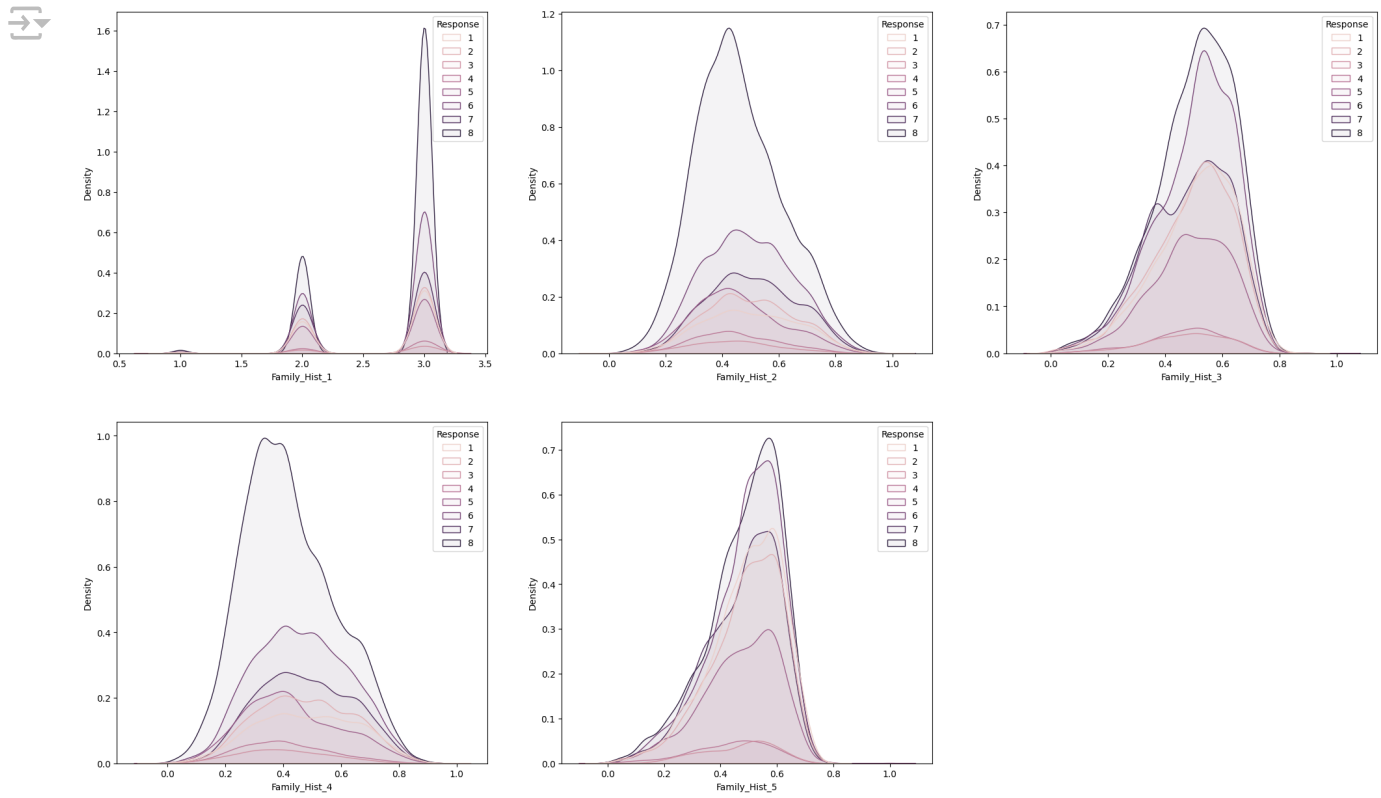
```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(25,20))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet5_InsuranceHistoryInfo].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])

# Delete any unused sets of axes in the subplot grid.
fig.delaxes(axes[2,2])
```



Insurance History Information

These KDE plots display various distributions with differing modalities across `Insurance_History` variables. However, a consistent trend is that they all closely overlap across each Response group/cohort of applicants, showing no significant differences in relative densities. Therefore, variations in these features are unlikely to individually contribute to predicting an applicant's risk rating.

```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(25,15))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet6_FamilyHistoryInfo].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])

# Delete any unused sets of axes in the subplot grid.
fig.delaxes(axes[1,2])
```



Family History Information

Family_Hist_1:

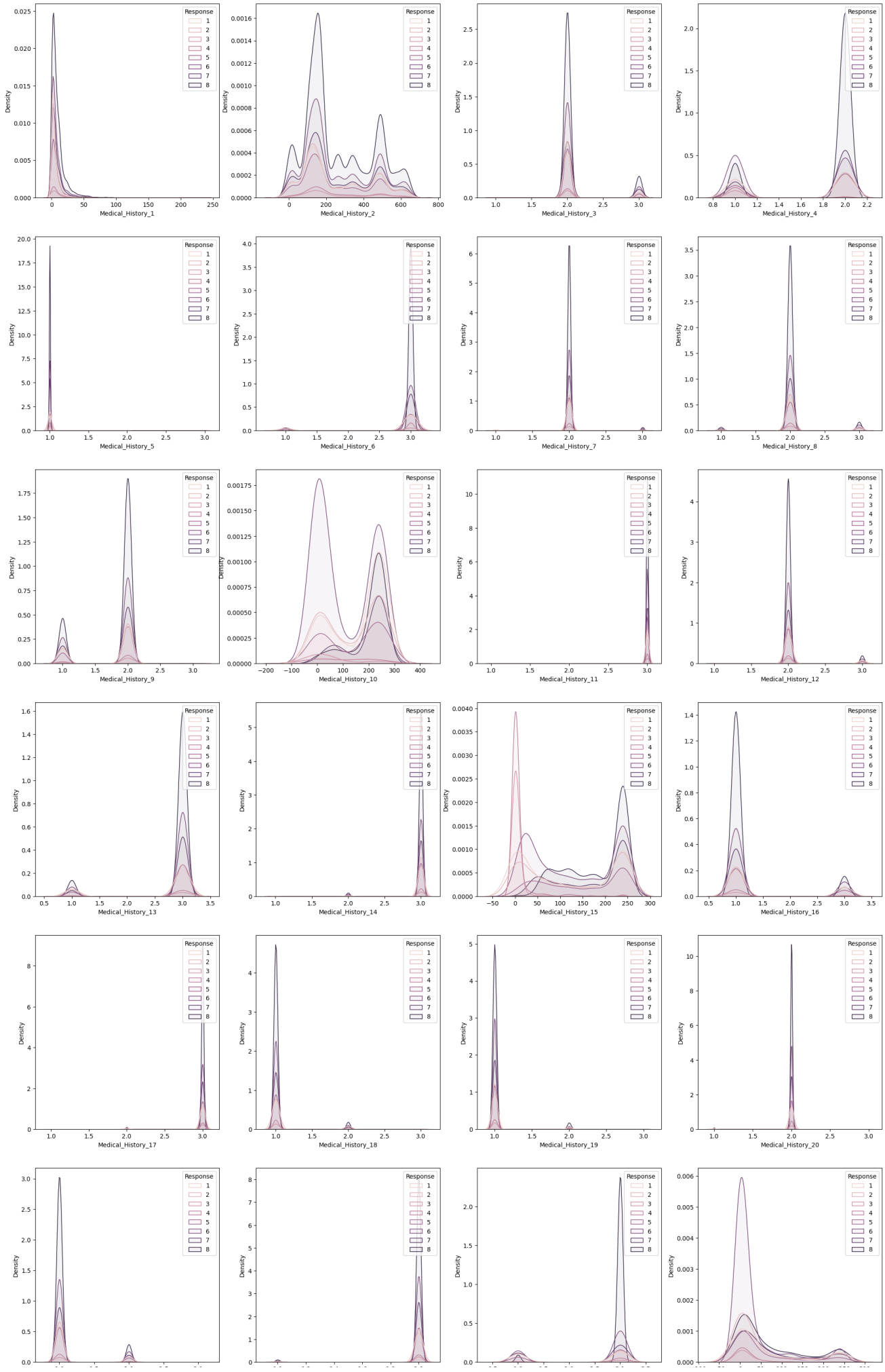
This KDE plot exhibits a bimodal distribution with peaks at x=2 and x=3, along with a very low-density curve at x=1. The most prominent peak is at x=3. However, since this feature shows minimal variation in relative densities across the different Response classes, it is unlikely that variations in this feature will significantly contribute to predicting an applicant's risk rating.

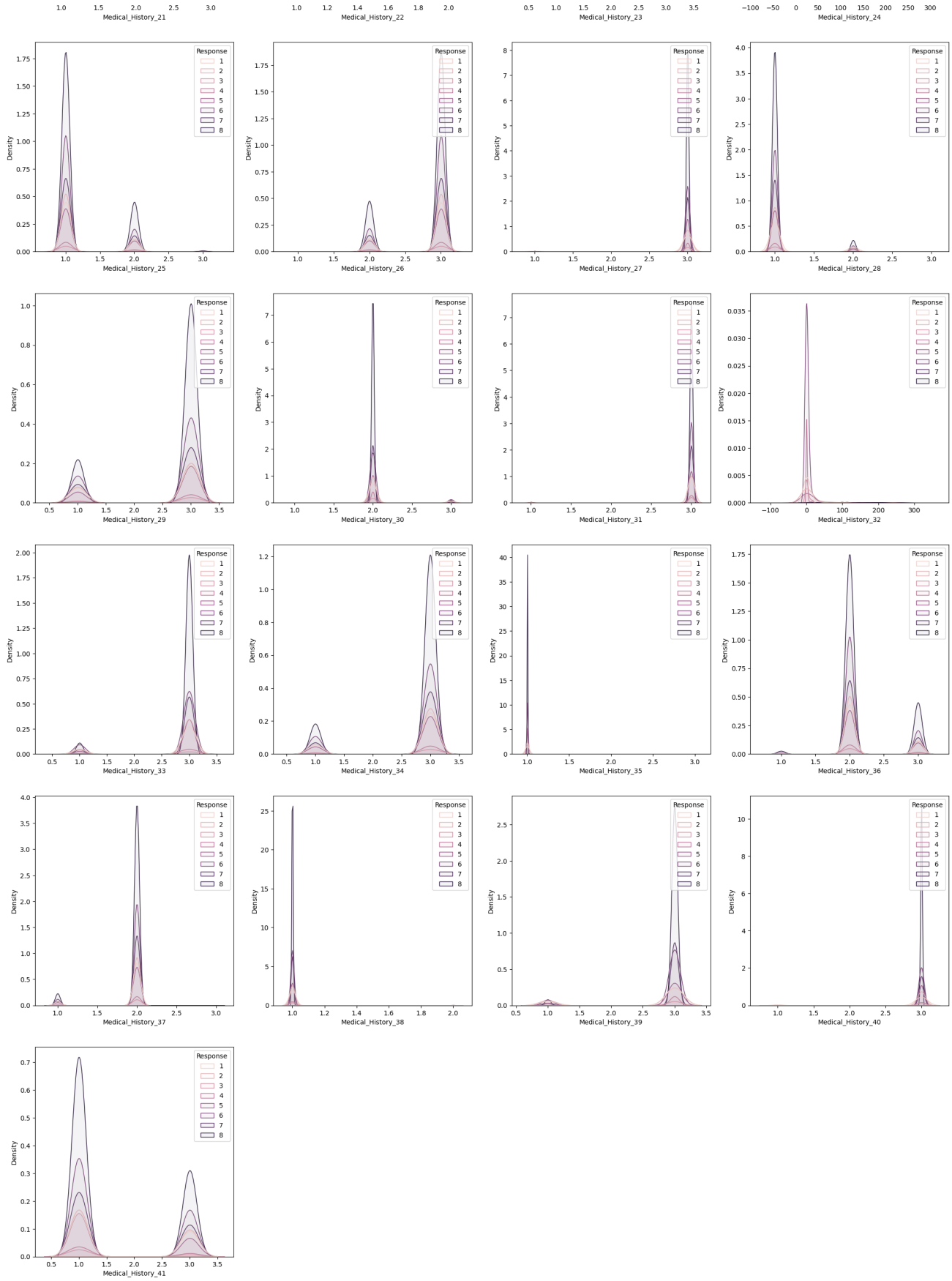
Family_Hist_2 - Family_Hist_5:

These KDE plots display several unimodal distributions with some variation in the composition and structure of each Response group's distribution. Each peak demonstrates broadening and shouldering. While most of the distributions' densities are spread between x=0.2 and x=0.8, each cohort's distribution varies somewhat in shape and skewness/kurtosis. For example, class 8 generally features more positive kurtosis compared to most other classes, which tend to show broader density plots.

```
# Set up a subplot grid.
fig, axes = plt.subplots(nrows=11, ncols=4, figsize=(25, 75))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet7_MedicalHistoryInfo].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//4,i%4])

# Delete any unused sets of axes in the subplot grid.
fig.delaxes(axes[10,1])
fig.delaxes(axes[10,2])
fig.delaxes(axes[10,3])
```

```
<ipython-input-23-523de3efe9db>:6: UserWarning: Dataset has 0 variance; skippi
  sns.kdeplot(data=df_index_set,
```




Medical_History_21
Medical_History_22
Medical_History_23
Medical_History_24
Medical_History_25
Medical_History_26
Medical_History_27
Medical_History_28
Medical_History_29
Medical_History_30
Medical_History_31
Medical_History_32
Medical_History_33
Medical_History_34
Medical_History_35
Medical_History_36
Medical_History_37
Medical_History_38
Medical_History_39
Medical_History_40
Medical_History_41
Density
Response
1
2
3
4
5
6
7
8

Medical History

Most of these KDE plots feature distributions that closely overlap across the different Response groups/cohorts of applicants. Therefore, variations in these underlying features are unlikely to individually provide predictive power for determining an applicant's risk rating. However, there are a few notable exceptions:

Medical_History_2/15/24:

These KDE plots show multimodal distributions with some degree of predictive distinction in terms of variance, as the peaks of each Response group distribution show different degrees of broadening and shouldering. However, it can be noted that the scales of the y-axes here, the densities for each underlying distribution, are very small, limiting the ability of these features to individually distinguish between Response groups.

Medical_History_10:

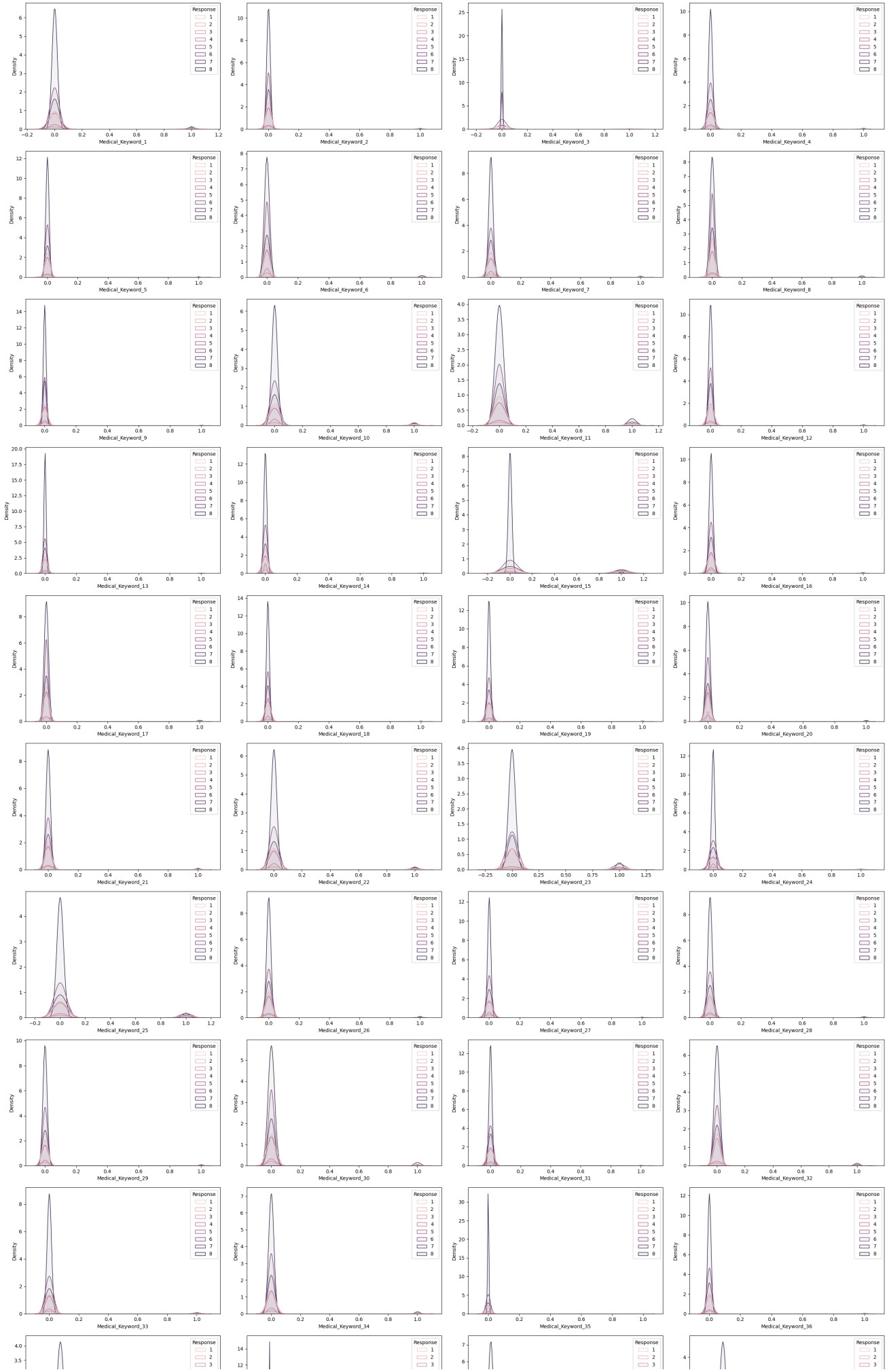
This KDE plot displays some interesting characteristics. For low-risk applicant cohorts, the distributions appear bimodal, whereas at higher risk levels, the distribution tends toward a single peak. However, as we later identify, this column has a very high proportion of missing values, therefore its distribution should not be misconstrued as highly predictive.

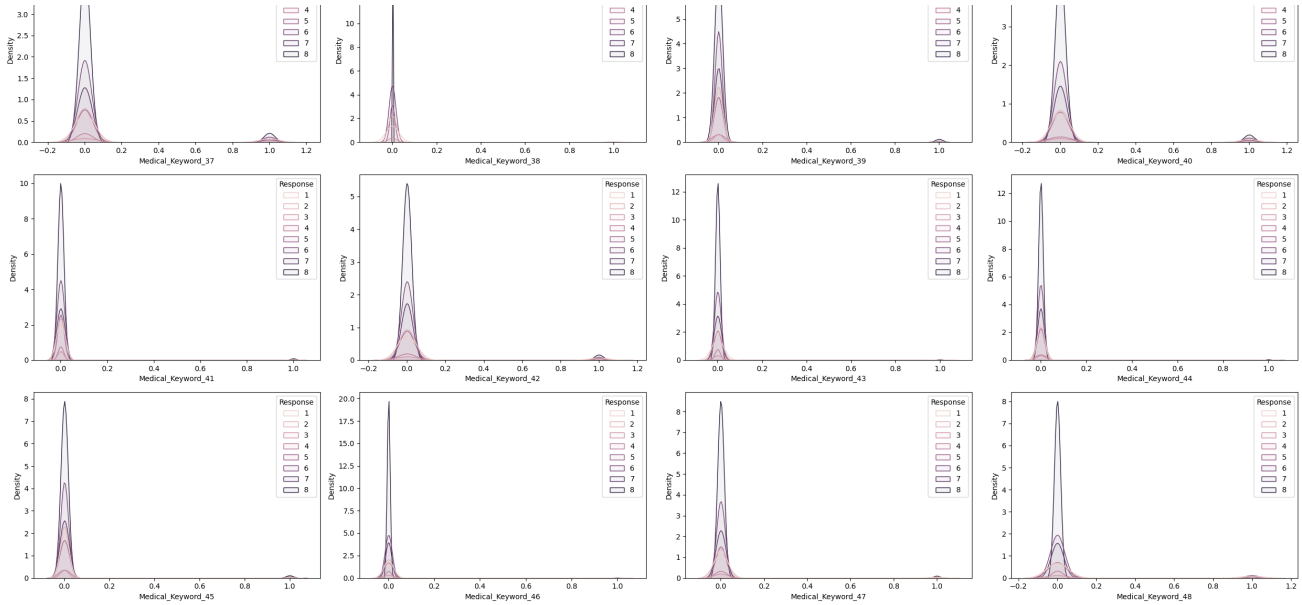
Medical_History_23:

This KDE plot indicates that this feature shows some potential. As the Response value/risk rating increases, each peak in the bimodal distribution becomes sharper, with a more positive kurtosis. In simpler terms, values further from the peaks' centers tend to correlate with a lower risk rating, whereas values that closely overlap with the peaks tend to represent applicants with higher risk ratings.

```
# Set up a subplot grid with a reduced figure size.
fig, axes = plt.subplots(nrows=12, ncols=4, figsize=(25,50))

# Produce kernel density estimate plots for each set of columns.
for i, column in enumerate(df_index_set[ColSet8_MedicalKeywordInfo].columns):
    sns.kdeplot(data=df_index_set,
                x=column,
                hue="Response", fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//4, i%4])

# Adjust layout to prevent overlap
plt.tight_layout()
plt.show()
```

```
<ipython-input-24-f103a7435cda>:6: UserWarning: Dataset has 0 variance; skippi
  sns.kdeplot(data=df_index_set,
```

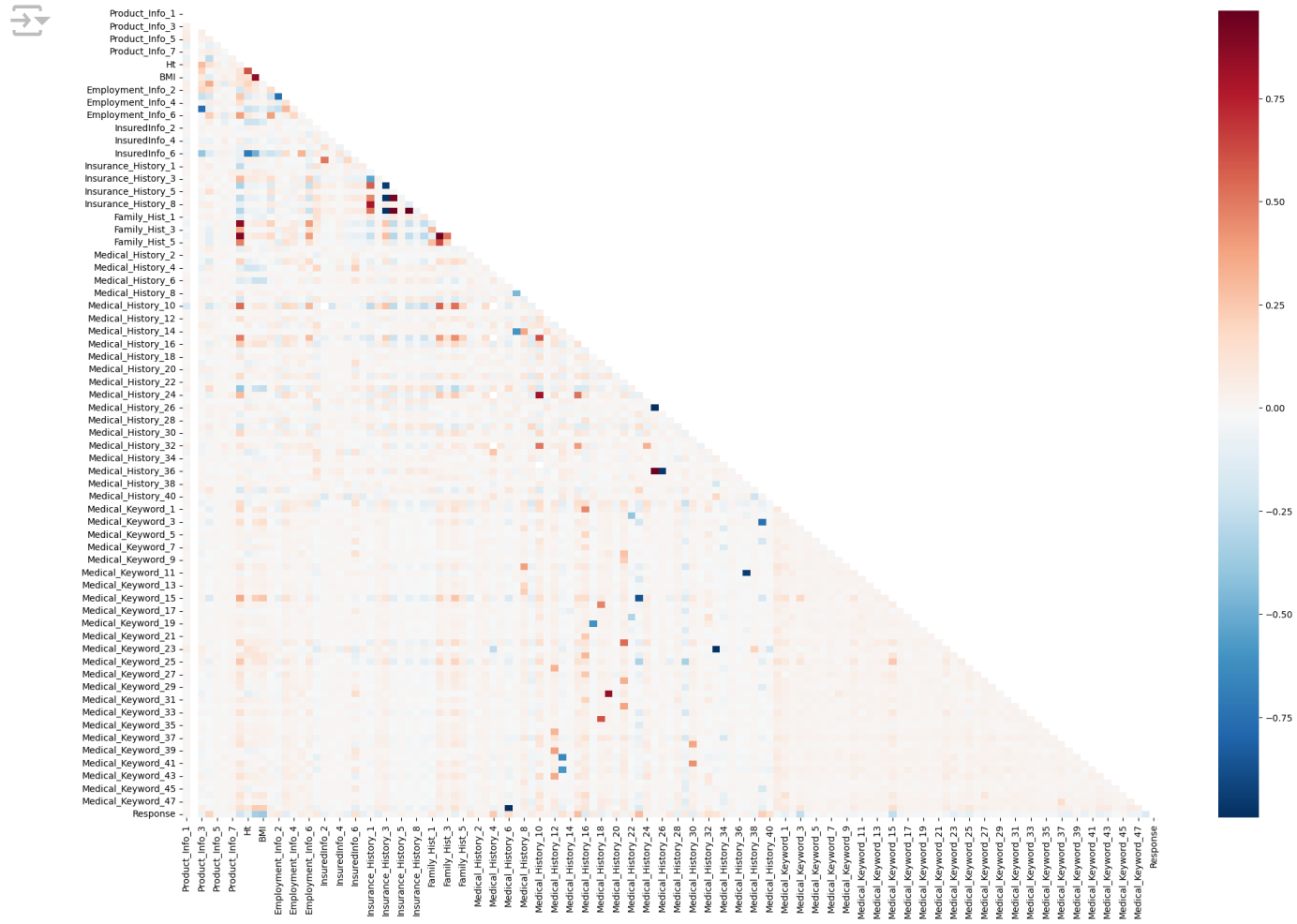



Density
Medical_Keyword_37
Medical_Keyword_38
Medical_Keyword_39
Medical_Keyword_40
Medical_Keyword_41
Medical_Keyword_42
Medical_Keyword_43
Medical_Keyword_44
Medical_Keyword_45
Medical_Keyword_46
Medical_Keyword_47
Medical_Keyword_48
Response
1
2
3
4
5
6
7
8

Medical Keywords

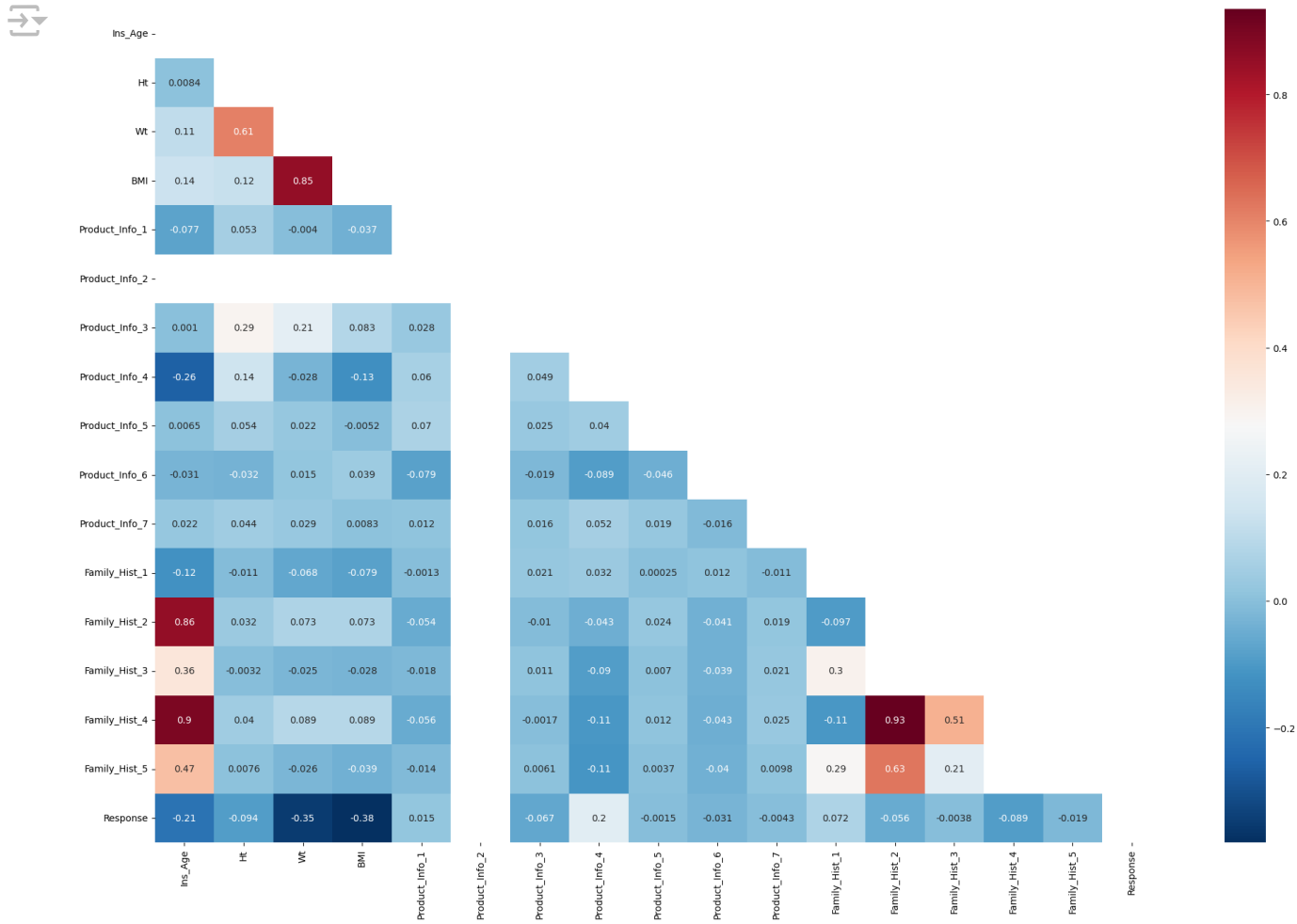
These KDE plots display distributions with varying modalities, yet they consistently show a close overlap between each Response group/cohort of applicants, with no significant differences in relative densities. Therefore, variations in these features are unlikely to individually contribute to predicting an applicant's risk rating.

```
# We know that at least columns in the Dataframe contain non-numeric data, prevent
# Convert relevant columns to numeric type if they contain numerical data represer
for column in df_index_set.columns:
    if df_index_set[column].dtype == 'object':  # Check if the column is of object
        try:
            df_index_set[column] = pd.to_numeric(df_index_set[column], errors='coe
        except:
            pass  # Skip columns that cannot be converted

# Produce a correlation matrix of the dataset - then, create a mask to hide the up
corrs = df_index_set.corr()
mask = np.zeros_like(corrs)
mask[np.triu_indices_from(mask)] = True

# Convert the correlation matrix into a heatmap using Seaborn.
plt.figure(figsize=(24,16))
sns.heatmap(corrs, cmap='RdBu_r', mask=mask)
plt.show()
```




```
# Assume df_index_set is your DataFrame

# Step 1: Specify the columns you want to include
columns_to_include = []  # Replace with your specific columns

# Add the predefined list to the columns_to_include list
columns_to_include.extend(ColSet2_ApplicantInfo)
columns_to_include.extend(ColSet1_ProdInfo)
columns_to_include.extend(ColSet6_FamilyHistoryInfo)
columns_to_include.append('Response')
```

```
# Step 2: Filter the DataFrame to include only the specified columns
df_filtered = df_index_set[columns_to_include]

# Step 3: Convert relevant columns to numeric type if they contain numerical data
for column in df_filtered.columns:
    if df_filtered[column].dtype == 'object':  # Check if the column is of object
        try:
            df_filtered[column] = pd.to_numeric(df_filtered[column], errors='coerc
        except:
            pass  # Skip columns that cannot be converted

# Step 4: Produce a correlation matrix of the filtered dataset
corrs = df_filtered.corr()

# Create a mask to hide the upper-right half of the matrix
mask = np.zeros_like(corrs)
mask[np.triu_indices_from(mask)] = True

# Convert the correlation matrix into a heatmap using Seaborn
plt.figure(figsize=(24, 16))
sns.heatmap(corrs, cmap='RdBu_r', mask=mask, annot=True)  # `annot=True` to show c
plt.show()
```



Based on the chart above, we can infer the following for each column set:

Column Set 1 - Product Info:

These features show minimal interaction or correlation with most other feature sets, except for `Employment_Info_1`, `Employment_Info_5`, and `Insured_Info_6`. These columns may be directly correlated because an applicant's employment or financial status can influence the type of policy or product they apply for.

Column Set 2 - Applicant Info:

These columns exhibit varying interactions with other feature sets, with the strongest correlations (excluding those within the same column set) observed between some `Family_Hist` columns and `Insured_Info_6`.

Column Set 3 - Employment Info:

Aside from two strong anti-correlations between `Employment_Info_2` and `Employment_Info_3`, and between `Employment_Info_5` and `Product_Info_3` and a few moderate interactions between `Employment_Info_6` and `Family_Hist_2` / `Family_Hist_4`, this column set does not strongly interact with other features.

Column Set 4 - Insured Info:

The column `InsuredInfo_2` shows a fairly strong correlation with `InsuredInfo_7` and a strong anti-correlation with some `Applicant Info` columns. Otherwise, this column set has minimal interaction with other features.

Column Set 5 - Insurance History Info:

This feature set exhibits several strong inter-correlations among the `Insurance_History` columns but shows minimal interaction with other features.

Column Set 6 - Family History Info:

The columns `Family_Hist_2` and `Family_Hist_4` show a very strong positive correlation with `Ins_Age`, and to a lesser degree with `Medical_History_10` and `Medical_History_15`.

Column Set 7 - Medical History Info:

This column set shows several correlation hotspots with various Medical_Keyword columns, as well as with `Ins_Age` and some `Family_Hist` columns.

Column Set 8 - Medical Keyword Info:

These columns exhibit several correlation hotspots with various `Medical_History` columns but do not show notable interactions with other features.

```
Start coding or generate with AI.
```

2 Data Pre-processing

2.1 Missing Values

It is important to assess the completeness of our dataset to ensure that any identified correlations or trends are supported by sufficient evidence. We need enough data points to validate any inferences or predictions derived from the dataset.

The following checks are designed to highlight areas where the dataset is incomplete.

```
# Determine which columns contain nulls/missing values.
cols_with_missing = [col for col in df_index_set.columns
                     if df_index_set[col].isnull().any()]

# Summarise how many missing values are present in each column.
df_index_set[cols_with_missing].isna().sum()
```

```
Product_Info_2         59381
Employment_Info_1         19
Employment_Info_4       6779
Employment_Info_6      10854
Insurance_History_5    25396
Family_Hist_2          28656
Family_Hist_3          34241
Family_Hist_4          19184
Family_Hist_5          41811
Medical_History_1       8889
Medical_History_10     58824
Medical_History_15     44596
Medical_History_24     55580
Medical_History_32     58274
dtype: int64
```

```
## Calculate the proportion of zeroes relative to non-zero values.
for col in cols_with_missing:
    sum = df_index_set[col].isna().sum()
    length = len(df_index_set[col].index)
    ratio = sum/length
    print('Proportion of zeroes in', col, 'is: ', round(ratio*100,2), '%.')
```

```
Proportion of zeroes in Product_Info_2 is:  100.0 %.
Proportion of zeroes in Employment_Info_1 is:  0.03 %.
Proportion of zeroes in Employment_Info_4 is:  11.42 %.
Proportion of zeroes in Employment_Info_6 is:  18.28 %.
Proportion of zeroes in Insurance_History_5 is:  42.77 %.
Proportion of zeroes in Family_Hist_2 is:  48.26 %.
Proportion of zeroes in Family_Hist_3 is:  57.66 %.
Proportion of zeroes in Family_Hist_4 is:  32.31 %.
Proportion of zeroes in Family_Hist_5 is:  70.41 %.
Proportion of zeroes in Medical_History_1 is:  14.97 %.
Proportion of zeroes in Medical_History_10 is:  99.06 %.
Proportion of zeroes in Medical_History_15 is:  75.1 %.
Proportion of zeroes in Medical_History_24 is:  93.6 %.
Proportion of zeroes in Medical_History_32 is:  98.14 %.
```

In the code above, we have examined the entire dataset to identify which columns contain missing values and to count the number of missing values in each of these columns.

Data Leakage

To prevent test-data leakage during model validation and testing, it is essential to ensure that only the training set is used to assess the completeness of our data. Therefore, we need to split the full dataset into training, validation, and testing sets. After splitting, we will review the training set for missing values.

```
Start coding or generate with AI.
```

2.2 Train-Test Split

Here, we will create separate DataFrames to store the features and target variables.

These DataFrames will then be used with the `train_test_split()` function from sklearn to divide the data into training, validation, and test subsets.

```
# Assign the features to their own dataframe.
X_original = df_index_set.drop(['Response'], axis=1)

# Assign the target variable to its own dataframe.
y_original = df_index_set.Response

# Perform a train-test split to obtain the training, validation and test data as s
from sklearn.model_selection import train_test_split

# Split out test/holdout set from full dataset.
# We will set the size of the X/y test datasets to be 20% of the original (full) X
X_other, X_test, y_other, y_test = train_test_split(X_original, y_original, train_

# Split remaining portion into training/validation sets.
# We will set the size of the X/y train datasets to be 60% of the original (full)
X_train, X_val, y_train, y_val = train_test_split(X_other, y_other, train_size=0.7
```

The dataset partitioning strategy involves allocating 20% of the original datasets for testing and 80% for training. Of the trsining data 0.25% is retained for validation.

2.3 Review and Handle Columns

2.3.1 Locating/handling excess zeroes

We now examine the training data to identify any missing values.

Statistical tests can determine if these values are missing completely at random (MCAR), missing at random (MAR), or missing not at random (MNAR). This assessment helps us understand whether the remaining data is still likely to represent the general population. For this analysis, for simplicity we assume that any missing data is MCAR and that any subsequent analysis or imputation is free from implicit bias.

We assume that the data is missing completely at random (MCAR) because, in many cases, missing values can occur due to non-systematic reasons, such as random errors in data entry or collection. This assumption allows us to proceed without complex adjustments. Alternatively, if we were not to assume MCAR, we would need to perform more rigorous statistical tests to determine the nature of the missing data (MAR or MNAR). This would involve understanding the underlying mechanisms causing the missing values and potentially using advanced imputation techniques or modeling approaches that account for the non-randomness in the data.

```
# Determine which columns contain nulls/missing values.
X_train_cols_with_missing = [col for col in X_train.columns if X_train[col].isnull
```

```
# Summarise how many missing values are present in each column.
X_train[X_train_cols_with_missing].isna().sum()
```

```
Product_Info_2         35628
Employment_Info_1         16
Employment_Info_4       4002
Employment_Info_6       6518
Insurance_History_5    15196
Family_Hist_2          17141
Family_Hist_3          20591
Family_Hist_4          11458
Family_Hist_5          25107
Medical_History_1       5368
Medical_History_10     35299
Medical_History_15     26744
Medical_History_24     33312
Medical_History_32     34980
dtype: int64
```

```
## Calculate the proportion of zeroes relative to non-zero values.
for col in X_train_cols_with_missing:
    sum = X_train[col].isna().sum()
    length = len(X_train[col].index)
    ratio = sum/length
    print('Proportion of zeroes in', col, 'is: ', round(ratio*100,2), '%.')
```

```
Proportion of zeroes in Product_Info_2 is:  100.0 %.
Proportion of zeroes in Employment_Info_1 is:  0.04 %.
Proportion of zeroes in Employment_Info_4 is:  11.23 %.
Proportion of zeroes in Employment_Info_6 is:  18.29 %.
Proportion of zeroes in Insurance_History_5 is:  42.65 %.
Proportion of zeroes in Family_Hist_2 is:  48.11 %.
Proportion of zeroes in Family_Hist_3 is:  57.79 %.
Proportion of zeroes in Family_Hist_4 is:  32.16 %.
Proportion of zeroes in Family_Hist_5 is:  70.47 %.
```

```
Proportion of zeroes in Medical_History_1 is:  15.07 %.
Proportion of zeroes in Medical_History_10 is:  99.08 %.
Proportion of zeroes in Medical_History_15 is:  75.06 %.
Proportion of zeroes in Medical_History_24 is:  93.5 %.
Proportion of zeroes in Medical_History_32 is:  98.18 %.
```

A number of columns have been identified in the code above, with different proportions of missing values - some are still in workable condition and can be preprocessed via imputation methods in order to provide machine-interpretable inputs for our models. However, there are still a handful of columns which are highly incomplete - any attempt to perform imputation would likely introduce significant bias/skew into these features' distributions.

Under these conditions, it may be safer to simply remove the columns altogether - intuitively, this makes sense as the number of non-blank values already represents a very small portion of these columns. Here, we have elected to delete columns where their proportions of missing values in the training subset are greater than 40%, selected as being as a reasonable threshold.

```
# These columns have been selected as they contain a high proportion of blanks/mis
cols_to_delete_due_to_missing_data = ['Insurance_History_5',
                                      'Family_Hist_2', 'Family_Hist_3', 'Family_Hi
                                      'Medical_History_10', 'Medical_History_15',

# Delete columns from ALL datasets where the proportion of zeroes in the TRAINING
X_train = X_train.drop(cols_to_delete_due_to_missing_data, axis=1)
X_val = X_val.drop(cols_to_delete_due_to_missing_data, axis=1)
X_test = X_test.drop(cols_to_delete_due_to_missing_data, axis=1)
```

Double-click (or enter) to edit

2.3.2 Handle Columns with Missing Values (Iterative Imputation)

In this step, we apply iterative imputation to the remaining columns with missing values. This process estimates the missing feature values based on the other features, effectively filling in the gaps with predicted values.

As a result, all rows and columns in the dataset will now contain machine-interpretable values, making the data ready for feature scaling, selection, and model fitting.

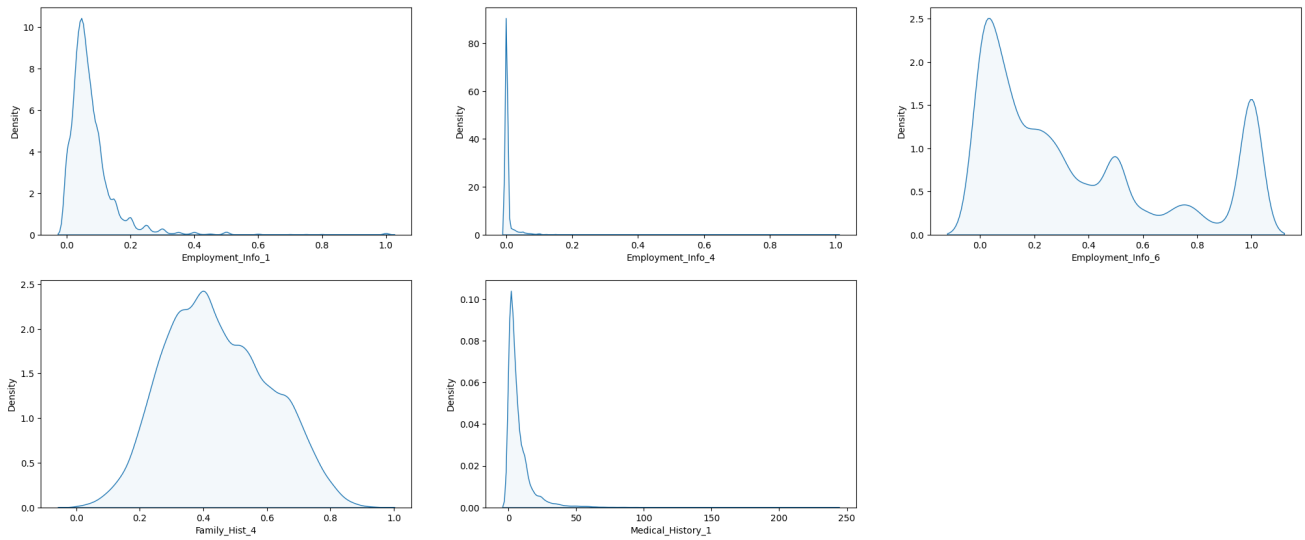
```
# These columns still contain missing values, and require imputation before they c
cols_to_impute = ['Employment_Info_1', 'Employment_Info_4', 'Employment_Info_6',
                  'Family_Hist_4',
                  'Medical_History_1']


# Import the IterativeImputer class from sklearn - NOTE: enable_iterative_imputer
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

```
# Take a copy of each dataset before transforming.
copy_X_train = X_train.copy()
copy_X_val = X_val.copy()
copy_X_test = X_test.copy()

# Filter the splits down to the columns that require imputation.
X_train_pre_impute = copy_X_train[cols_to_impute]
X_val_pre_impute = copy_X_val[cols_to_impute]
X_test_pre_impute = copy_X_test[cols_to_impute]

# Save the other columns into separate dataframes, for re-joining later on.
X_train_no_impute = copy_X_train.drop(cols_to_impute, axis=1)
X_val_no_impute = copy_X_val.drop(cols_to_impute, axis=1)
X_test_no_impute = copy_X_test.drop(cols_to_impute, axis=1)

# Initialise the IterativeImputer transformer.
X_imputer = IterativeImputer(random_state=0)

# Transform the train/val/test datasets using iterative imputation.
X_train_post_impute = pd.DataFrame(X_imputer.fit_transform(X_train_pre_impute), co
X_val_post_impute = pd.DataFrame(X_imputer.transform(X_val_pre_impute), columns=X_
X_test_post_impute = pd.DataFrame(X_imputer.transform(X_test_pre_impute), columns=

# Reset the indexes of each dataset, as they are dropped during imputation.
X_train_post_impute.index = X_train_pre_impute.index
X_val_post_impute.index = X_val_pre_impute.index
X_test_post_impute.index = X_test_pre_impute.index

# Re-join the imputed columns with the remaining columns in each dataset.
X_train_imputed = pd.concat([X_train_no_impute, X_train_post_impute], axis=1)
X_val_imputed = pd.concat([X_val_no_impute, X_val_post_impute], axis=1)
X_test_imputed = pd.concat([X_test_no_impute, X_test_post_impute], axis=1)
```

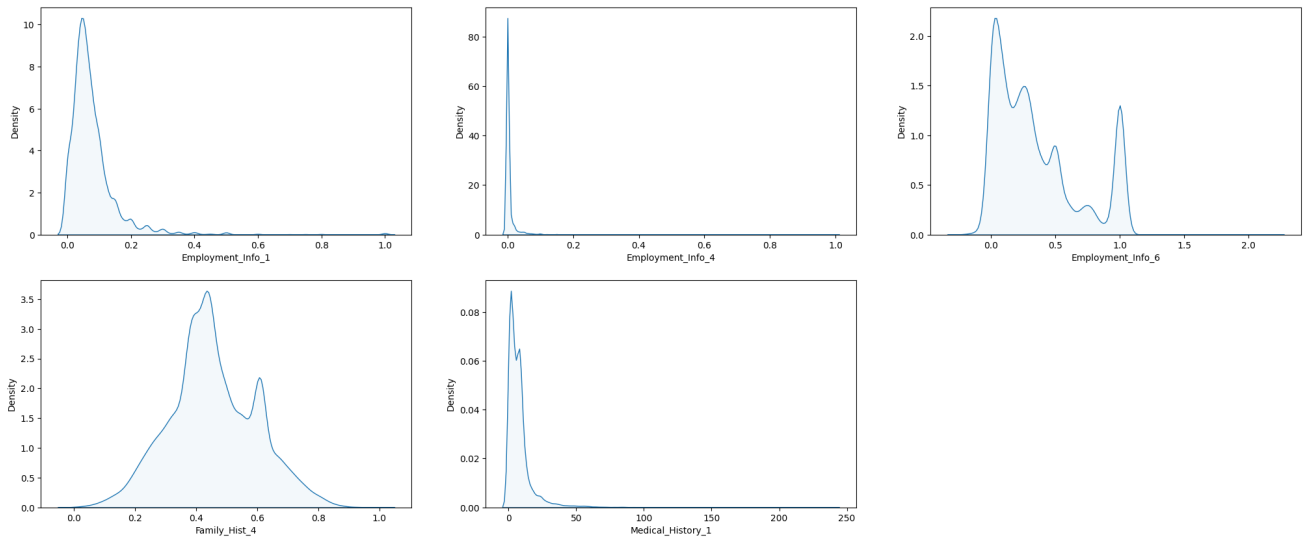
2.3.3 Checks - before/after iterative imputation

We will now review the dataset before and after iterative imputation, in order to understand how each column's distribution has been affected.

```
# KDE Plots - Before imputation.

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(25,10))

for i, column in enumerate(df_index_set[X_train_post_impute.columns].columns):
    sns.kdeplot(data=df_index_set[X_train_post_impute.columns],
                x=column,
                fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])

fig.delaxes(axes[1,2])
```




```
# KDE Plots - After imputation.

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(25,10))

for i, column in enumerate(X_train_post_impute.columns):
    sns.kdeplot(data=X_train_post_impute,
                x=column,
                fill=True, common_norm=True, alpha=0.05,
                ax=axes[i//3,i%3])

fig.delaxes(axes[1,2])
```



As shown above, the distributions of three of the five columns remain largely unchanged. However, we have introduced additional probability density and peak splitting into `Family_Hist_4` (around x=0.4 and x=0.6) and `Medical_History_1` (around x=10).

It is important to be cautious later when evaluating the significance of these two features and explaining how predictions were generated for our chosen model. This is crucial because the changes in these distributions might influence how the model interprets and uses these features, potentially affecting the model's accuracy and reliability.

2.4 Review potential sources of data leakage

Data leakage or "data snooping" occurs when information from outside the training dataset is included in creating the model. We need to ensure that no aspect of data preprocessing, feature selection, or model evaluation is influenced by data that should not be known at the point of training. Examples of this includes data not available at the time of prediction and features based on the entire dataset, including the test set, rather than just the training set.

In this case the Response variable could artificially boost model performance and would not be indicative of real-world performance. Applying transformations such as scaling and normalization to the entire dataset before splitting would causes information leakage from the test set into the training process.

Other examples are when cross-validation folds are not properly managed, allowing data from the test fold to influence the training process, which can occur if time series data is randomly split without considering temporal order.Also including features directly correlated with the target variable but not available at the time of predictionc e.g. any variable that is a direct result of the target variable.

Impact of Data Snooping

Impacts of data snooping would be that the above would results in the model appearing to perform much better on the test data than it would on unseen data. The model may not generalize well to new, unseen data because it has been influenced by information it wouldn't have in a real-world scenario.

Avoiding Data Snooping

To avoid data snooping for this data set we take the following steps;

Data Splitting

We have split the data using train_test_split() into training, validation and test sets before any preprocessing steps.

Separate Data Processing

We apply data transformations within the training set only, and then apply the same transformations to the validation and test sets.

Feature Engineering

Generally we should perform feature engineering only on the training set and then apply the engineered features to the test set without recalculating.

Cross-Validation Practices

In general, we are using cross-validation techniques that respect the structure and any temporal aspects of the data (however we are not using timeseries data).

Carefully managing data preparation to ensure strict separation between training and evaluation phases, data snooping can be effectively avoided, leading to more accurate and reliable models.

Double-click (or enter) to edit

3. Feature Engineering

3.1 Perform feature engineering (using training data)

In this section, we implement supervised (e.g. categorical encoding) and unsupervised (e.g. K-means clustering) learning techniques in order to create new features within our dataset that our models can be fitted to.

```
# Create a clone copy of each imputed dataset to avoid changing any original data.
copy_X_train_imputed = X_train_imputed.copy()
copy_X_val_imputed = X_val_imputed.copy()
copy_X_test_imputed = X_test_imputed.copy()
```

3.1.1 One-Hot Encoding

We will one-hot encode the `Product_Info_2` column, to convert any categorical inputs are to numerical, machine-interpretable values that can be supplied to each classification model.

```
from sklearn.preprocessing import OneHotEncoder

# Initialise a one-hot encoder to columns that contain categorical data.
OH_encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
OH_col = ['Product_Info_2']

## We set handle_unknown='ignore' to avoid errors when the validation data contair
## in the training data, setting sparse=False ensures that the encoded columns are
## (instead of a sparse matrix).

# Use the one-hot encoder to transform the categorical data columns.
OH_col_train = pd.DataFrame(OH_encoder.fit_transform(copy_X_train_imputed[OH_col]))
OH_col_val = pd.DataFrame(OH_encoder.transform(copy_X_val_imputed[OH_col]))
OH_col_test = pd.DataFrame(OH_encoder.transform(copy_X_test_imputed[OH_col]))

# One-hot encoding removes the index; re-assign the original index.
OH_col_train.index = copy_X_train_imputed.index
OH_col_val.index = copy_X_val_imputed.index
OH_col_test.index = copy_X_test_imputed.index

# Add column-labelling back in, using the get_feature_names_out() function.
OH_col_train.columns = OH_encoder.get_feature_names_out(OH_col)
OH_col_val.columns = OH_encoder.get_feature_names_out(OH_col)
OH_col_test.columns = OH_encoder.get_feature_names_out(OH_col)

# Create dataframes that only include the numerical features/columns (these will k
copy_X_train_imputed_no_OH_col = copy_X_train_imputed.drop(OH_col, axis=1)
copy_X_val_imputed_no_OH_col = copy_X_val_imputed.drop(OH_col, axis=1)
copy_X_test_imputed_no_OH_col = copy_X_test_imputed.drop(OH_col, axis=1)

# Concatenate the one-hot encoded columns with the existing numerical features/col
X_train_enc = pd.concat([copy_X_train_imputed_no_OH_col, OH_col_train], axis=1)
X_val_enc = pd.concat([copy_X_val_imputed_no_OH_col, OH_col_val], axis=1)
X_test_enc = pd.concat([copy_X_test_imputed_no_OH_col, OH_col_test], axis=1)
```

3.1.2 Scaling / Normalisation

Here, we perform min-max scaling on the encoded datasets, so that all features lie between 0 and 1 - so that, when training any of the classification models, all features will have variances with the same order of magnitude as each other. Thus, no single feature will dominate the objective function and prohibit the model from learning from other features correctly as expected.

```
from sklearn.preprocessing import MinMaxScaler

# Initialise the MinMaxScaler model, then fit it to the (encoded) training dataset
MM_scaler = MinMaxScaler()
MM_scaler.fit(X_train_enc)

# Then, normalise/transform the training, validation and test datasets.
X_train_scale = pd.DataFrame(MM_scaler.transform(X_train_enc),
                             index=X_train_enc.index,
                             columns=X_train_enc.columns)

X_val_scale = pd.DataFrame(MM_scaler.transform(X_val_enc),
                             index=X_val_enc.index,
                             columns=X_val_enc.columns)

X_test_scale = pd.DataFrame(MM_scaler.transform(X_test_enc),
                             index=X_test_enc.index,
                             columns=X_test_enc.columns)
```

3.1.3 K-means clustering

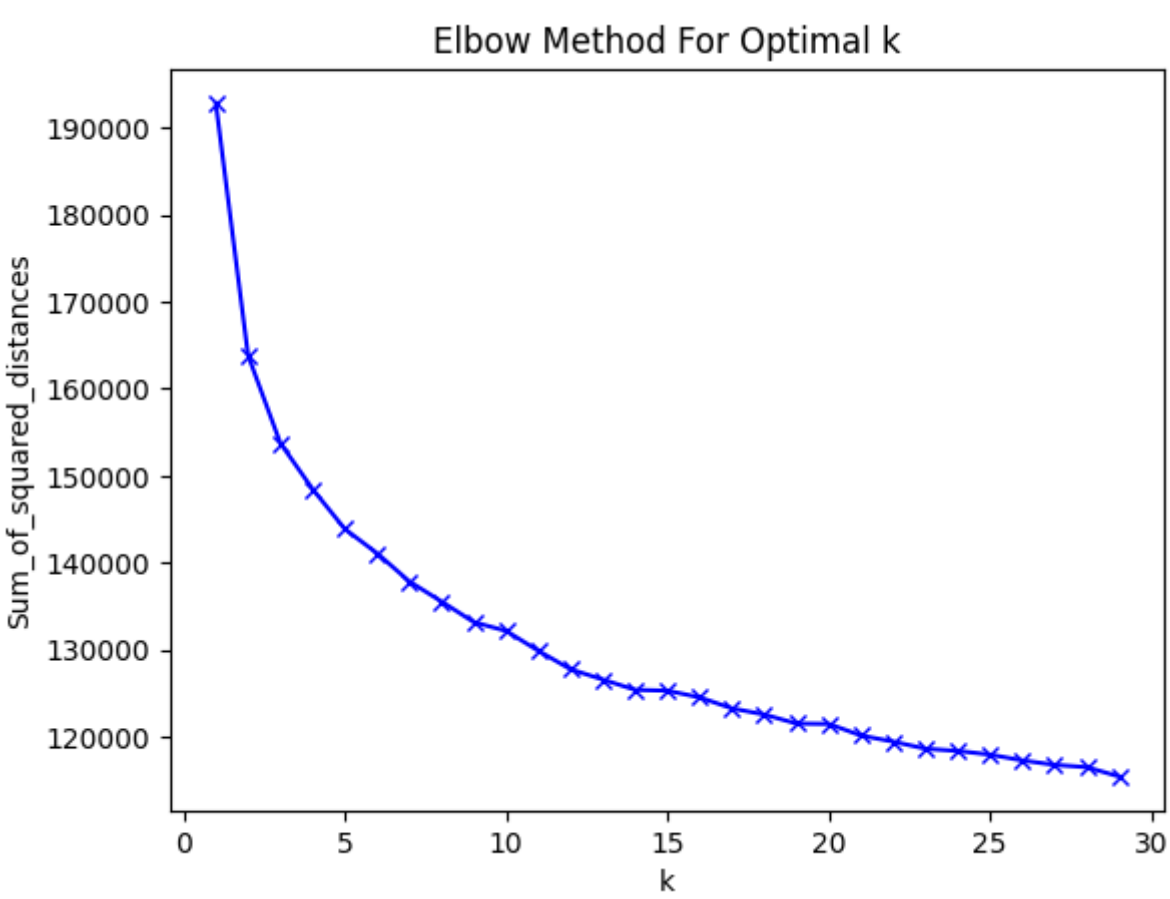
We now use K-means clustering to group together applicants based on their commonalities. First we will train our K-means clustering algorithm using the training subset, before predicting cluster groups on all three subsets.

These cluster labels will be incorporated as an additional feature in our datasets, and may in fact prove to be useful in helping to understand applicants' risk rating assignments later on.

```
# Create copies of the scaled datasets, prior to performing K-Means clustering.
copy_X_train_scale = X_train_scale.copy()
copy_X_val_scale = X_val_scale.copy()
copy_X_test_scale = X_test_scale.copy()

from sklearn.cluster import KMeans
```

We now use the elbow-analysis method to determine the optimal number of clusters to use.

```
# Determine the optimal number of clusters.
# Method: Cluster the dataset into k clusters, then calculate the inertia/sum of s
# Repeat this by looping through k=1 to k=30.

Sum_of_squared_distances = []
K = range(1,30)
for k in K:
    km = KMeans(n_clusters=k, n_init=10)
    km = km.fit(copy_X_train_scale)
    Sum_of_squared_distances.append(km.inertia_)


# Create a plot of K-values versus their respective inertias/sums of squared dista
plt.plot(K, Sum_of_squared_distances, 'bx-')
plt.xlabel('k')
plt.ylabel('Sum_of_squared_distances')
plt.title('Elbow Method For Optimal k')
plt.show()
```



As shown above, the "elbow" of the curve begins to form at k=15.

Increasing k beyond this value does not yield a significant benefit in the rate of reduction in the training dataset's inertia - so, we will set k=15 when initialising the KMeans() clustering algorithm below.

```
# Set n_clusters=15, as derived from the elbow-method analysis above:
kmeans = KMeans(n_clusters=15, n_init=10, random_state=0)
```

```
# Fit the K-Means clustering algorithm to the training dataset, then predict the t
copy_X_train_scale["Cluster"] = kmeans.fit_predict(copy_X_train_scale)
copy_X_val_scale["Cluster"] = kmeans.predict(copy_X_val_scale)
copy_X_test_scale["Cluster"] = kmeans.predict(copy_X_test_scale)

# Convert the cluster labels into one-hot encoded variants.
X_train_cluster_OH_enc = pd.get_dummies(copy_X_train_scale.Cluster).add_prefix('KM
X_val_cluster_OH_enc = pd.get_dummies(copy_X_val_scale.Cluster).add_prefix('KMeans
X_test_cluster_OH_enc = pd.get_dummies(copy_X_test_scale.Cluster).add_prefix('KMea

# Re-join the K-Means clustering labels onto the original dataframes.
X_train_KMeans = pd.concat([copy_X_train_scale, X_train_cluster_OH_enc], axis=1)
X_val_KMeans = pd.concat([copy_X_val_scale, X_val_cluster_OH_enc], axis=1)
X_test_KMeans = pd.concat([copy_X_test_scale, X_test_cluster_OH_enc], axis=1)

# Remove the initially derived "Cluster" columns from each dataset.
X_train_KMeans = X_train_KMeans.drop(['Cluster'], axis=1)
X_val_KMeans = X_val_KMeans.drop(['Cluster'], axis=1)
X_test_KMeans = X_test_KMeans.drop(['Cluster'], axis=1)
```

3.2 Perform feature selection (using validation data)

In this section, we use several methods to determine which features within our dataset will be most important for fitting each of the classification models - this is done in order to prevent overfitting.

At this stage, it is important to now switch over to using the validation dataset for feature selection/model refinement, so that we do not continually rely on the training dataset and risk invoking data leakage into our model generation process.

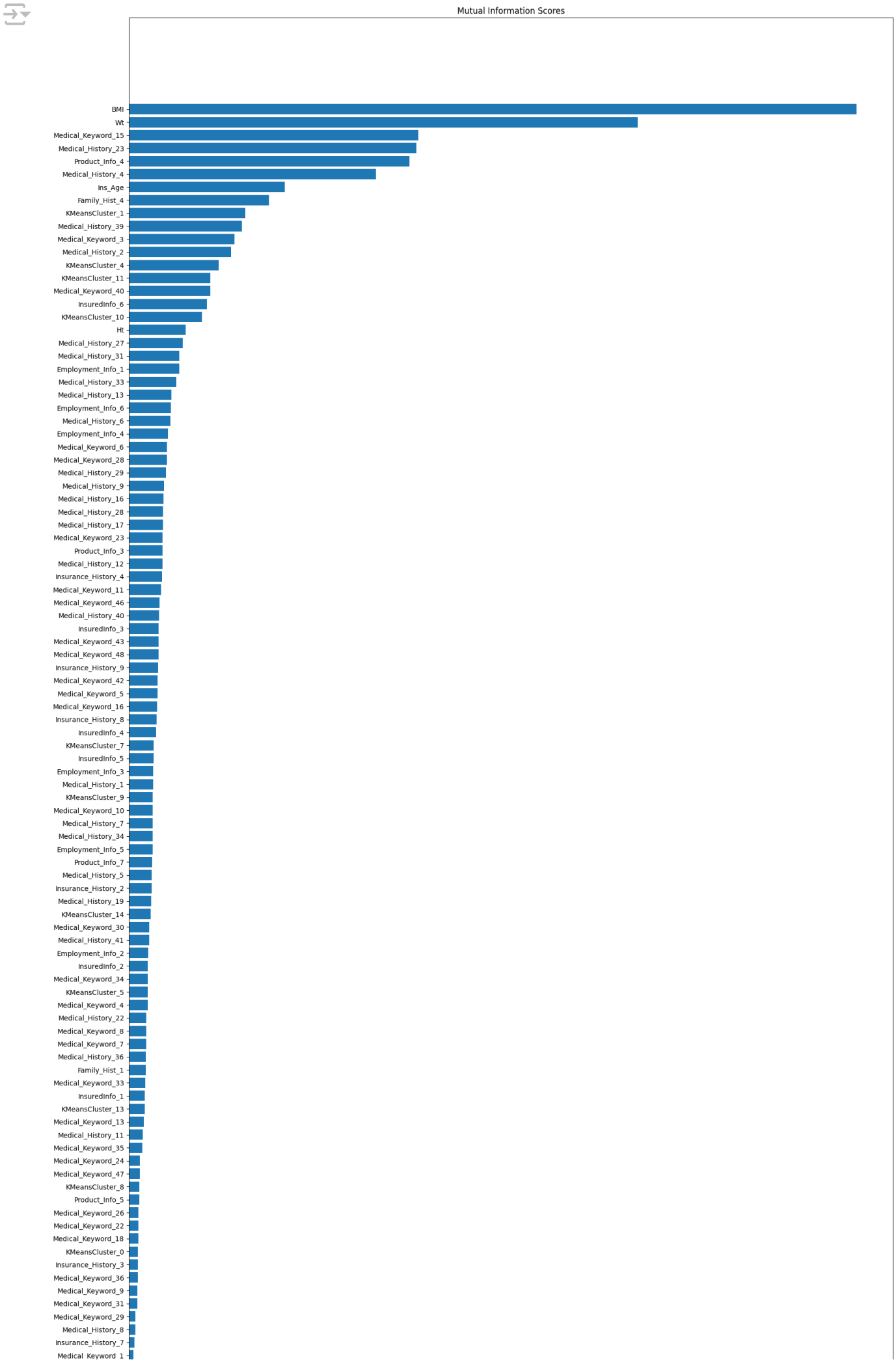
3.2.1 Method 1: Mutual Information (MI)

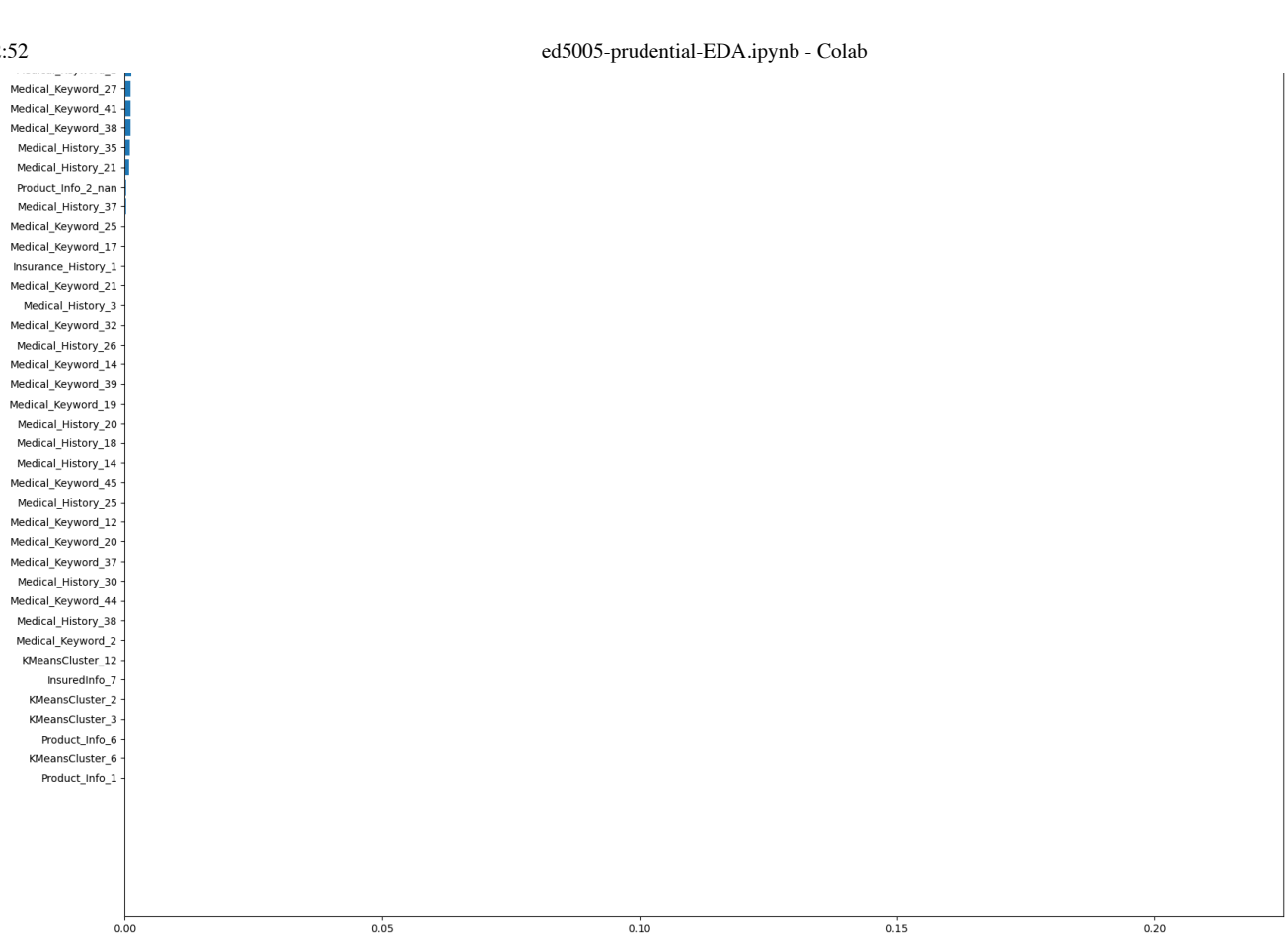
Here, we will calculate the MI scores of the validation dataset, using two custom functions that rely on the mutual_info_classif() function available within sklearn.

This will help us to understand whether there are any useful features in our dataset that should be preserved, during feature selection.

```
# Import the mutual_info_classif() class from sklearn.
from sklearn.feature_selection import mutual_info_classif

# Define a custom function that calculates Mutual Information (MI) scores for a gi
def make_mi_scores(X, y):
    mi_scores = mutual_info_classif(X, y)
    mi_scores = pd.Series(mi_scores, name="MI Scores", index=X.columns)
    mi_scores = mi_scores.sort_values(ascending=False)
```

```
    return mi_scores

# Define a custom function that plots MI scores in descending order (i.e. most imp
def plot_mi_scores(scores):
    scores = scores.sort_values(ascending=True)
    width = np.arange(len(scores))
    ticks = list(scores.index)
    plt.barh(width, scores)
    plt.yticks(width, ticks)
    plt.title("Mutual Information Scores")


# Calculate MI scores on the validation dataset.
mi_scores_X_val = make_mi_scores(X_val_KMeans, y_val)

# Plot the MI scores obtained from the validation dataset.
plt.figure(dpi=100, figsize=(20,50))
plot_mi_scores(mi_scores_X_val)
```


Mutual Information Scores
BMI
Wt
Medical_Keyword_15
Medical_History_23
Product_Info_4
Medical_History_4
Ins_Age
Family_Hist_4
KMeansCluster_1
Medical_History_39
Medical_Keyword_3
Medical_History_2
KMeansCluster_4
KMeansCluster_11
Medical_Keyword_40
InsuredInfo_6
KMeansCluster_10
Ht
Medical_History_27
Medical_History_31
Employment_Info_1
Medical_History_33
Medical_History_13
Employment_Info_6
Medical_History_6
Employment_Info_4
Medical_Keyword_6
Medical_Keyword_28
Medical_History_29
Medical_History_9
Medical_History_16
Medical_History_28
Medical_History_17
Medical_Keyword_23
Product_Info_3
Medical_History_12
Insurance_History_4
Medical_Keyword_11
Medical_Keyword_46
Medical_History_40
InsuredInfo_3
Medical_Keyword_43
Medical_Keyword_48
Insurance_History_9
Medical_Keyword_42
Medical_Keyword_5
Medical_Keyword_16
Insurance_History_8
InsuredInfo_4
KMeansCluster_7
InsuredInfo_5
Employment_Info_3
Medical_History_1
KMeansCluster_9
Medical_Keyword_10
Medical_History_7
Medical_History_34
Employment_Info_5
Product_Info_7
Medical_History_5
Insurance_History_2
Medical_History_19
KMeansCluster_14
Medical_Keyword_30
Medical_History_41
Employment_Info_2
InsuredInfo_2
Medical_Keyword_34
KMeansCluster_5
Medical_Keyword_4
Medical_History_22
Medical_Keyword_8
Medical_Keyword_7
Medical_History_36
Family_Hist_1
Medical_Keyword_33
InsuredInfo_1
KMeansCluster_13
Medical_Keyword_13
Medical_History_11
Medical_Keyword_35
Medical_Keyword_24
Medical_Keyword_47
KMeansCluster_8
Product_Info_5
Medical_Keyword_26
Medical_Keyword_22
Medical_Keyword_18
KMeansCluster_0
Insurance_History_3
Medical_Keyword_36
Medical_Keyword_9
Medical_Keyword_31
Medical_Keyword_29
Medical_History_8
Insurance_History_7
Medical_Keyword_1


Medical_Keyword_27
Medical_Keyword_41
Medical_Keyword_38
Medical_History_35
Medical_History_21
Product_Info_2_nan
Medical_History_37
Medical_Keyword_25
Medical_Keyword_17
Insurance_History_1
Medical_Keyword_21
Medical_History_3
Medical_Keyword_32
Medical_History_26
Medical_Keyword_14
Medical_Keyword_39
Medical_Keyword_19
Medical_History_20
Medical_History_18
Medical_History_14
Medical_Keyword_45
Medical_History_25
Medical_Keyword_12
Medical_Keyword_20
Medical_Keyword_37
Medical_History_30
Medical_Keyword_44
Medical_History_38
Medical_Keyword_2
KMeansCluster_12
InsuredInfo_7
KMeansCluster_2
KMeansCluster_3
Product_Info_6
KMeansCluster_6
Product_Info_1
0.00
0.05
0.10
0.15
0.20

From the chart above, we can see that the top 5 ranked features (in descending order) are BMI, Wt, Product_Info_4, Medical_Keyword_15, and Medical_History_23. This means that these features have strong statistical dependences with the Response variable, i.e. that they contribute significantly to reducing uncertainty in the value of the Response variable (given a known value of the feature).

Features that have low/zero MI scores indicate that they do not significantly contribute towards reducing this uncertainty, and are hence less useful for guiding our predictions.

3.2.2 Method 2: Multicollinearity analysis

We will now review our dataset via Variance Inflation Factor analysis, which is critical for detecting the presence of multicollinearity (i.e. where several independent variables in a model are highly correlated - hence resulting in less reliable statistical inferences).

This will help us to understand whether there are any redundant features in our dataset that should not be kept, during feature selection.

```
# Import variance_inflation_factor from statsmodels
from statsmodels.stats.outliers_influence import variance_inflation_factor

# Define a custom function that calculates variance inflation factor (VIF) scores
def calc_vif(X):
    vif = pd.DataFrame()
    vif["Variables"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1]
    return(vif)

# Select only numeric columns before calculating VIF
# Ensures that only numeric columns are passed to the calc_vif function, resolving
```

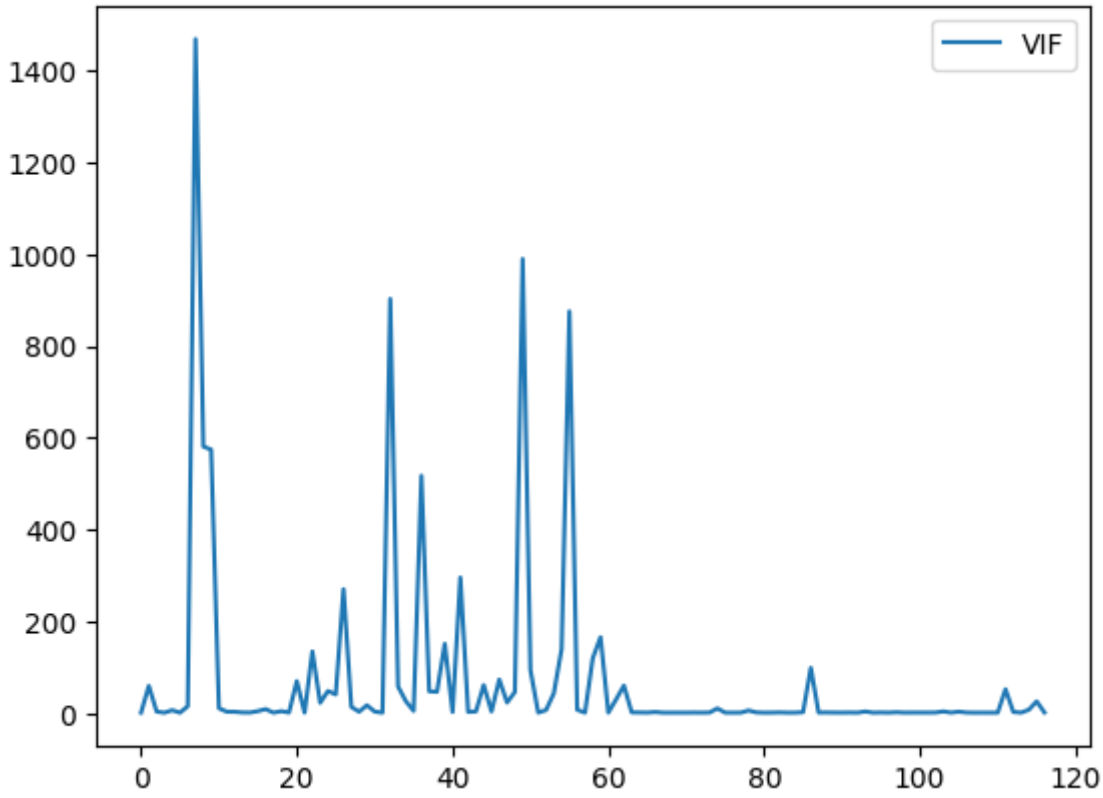
```
X_val_KMeans_numeric = X_val_KMeans.select_dtypes(include=[np.number])

# Calculate VIF scores on the numeric columns of the validation dataset.
vif_scores = calc_vif(X_val_KMeans_numeric)
# "RuntimeWarning: divide by zero" can be safely ignored as this is caused by perf
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/regression/linear_model.py
  return 1 - self.ssr/self.uncentered_tss
```

```
# Plot the VIF scores obtained from the validation dataset.
vif_scores.plot()

# Display all columns with VIF scores > 10.
vif_scores.loc[vif_scores['VIF'] > 10]
```

	Variables	VIF
1	Product_Info_3	59.419387
6	Ins_Age	15.199446
7	Ht	1468.410060
8	Wt	580.891692
9	BMI	574.150018
10	Employment_Info_2	10.707951
20	Insurance_History_1	69.910903
22	Insurance_History_3	134.378070
23	Insurance_History_4	22.562908
24	Insurance_History_7	48.202031
25	Insurance_History_8	40.892993
26	Insurance_History_9	270.113400
27	Family_Hist_1	13.659679
29	Medical_History_3	17.201201
32	Medical_History_6	902.477716
33	Medical_History_7	58.197347
34	Medical_History_8	26.104757
36	Medical_History_11	517.173109
37	Medical_History_12	47.592428
38	Medical_History_13	46.566427
39	Medical_History_14	151.063000
41	Medical_History_17	295.523219
44	Medical_History_20	61.270406
46	Medical_History_22	73.331654
47	Medical_History_23	23.133341
48	Medical_History_25	46.184935
49	Medical_History_26	989.508944
50	Medical_History_27	93.668434
53	Medical_History_30	43.152358
54	Medical_History_31	139.521735
55	Medical_History_33	875.322051
58	Medical_History_36	119.586731

59	Medical_History_37	165.544548
61	Medical_History_39	30.016709
62	Medical_History_40	60.073108
74	Medical_Keyword_11	10.333894
86	Medical_Keyword_23	98.798750
111	Medical_Keyword_48	52.167161
115	Family_Hist_4	25.742316



The features listed above have very high VIF scores, which indicate a high level of multicollinearity. However, in the edge cases where some of these values tend towards infinity, these can be discounted as they represent dummy variables that are perfectly anti-correlated (e.g. one applicant/row in the dataset can only belong to a single K-Means cluster).

More information on when it is safe to ignore multicollinearity can be found here: Multicollinearity - Statistical Horizons.

3.2.3 Method 3: Principal Component Analysis

Next, we will perform Principal Component Analysis in order to understand the most significant sources of variation within our dataset.

This will help us to understand whether there are particularly useful features in our dataset that should be preserved, during feature selection.

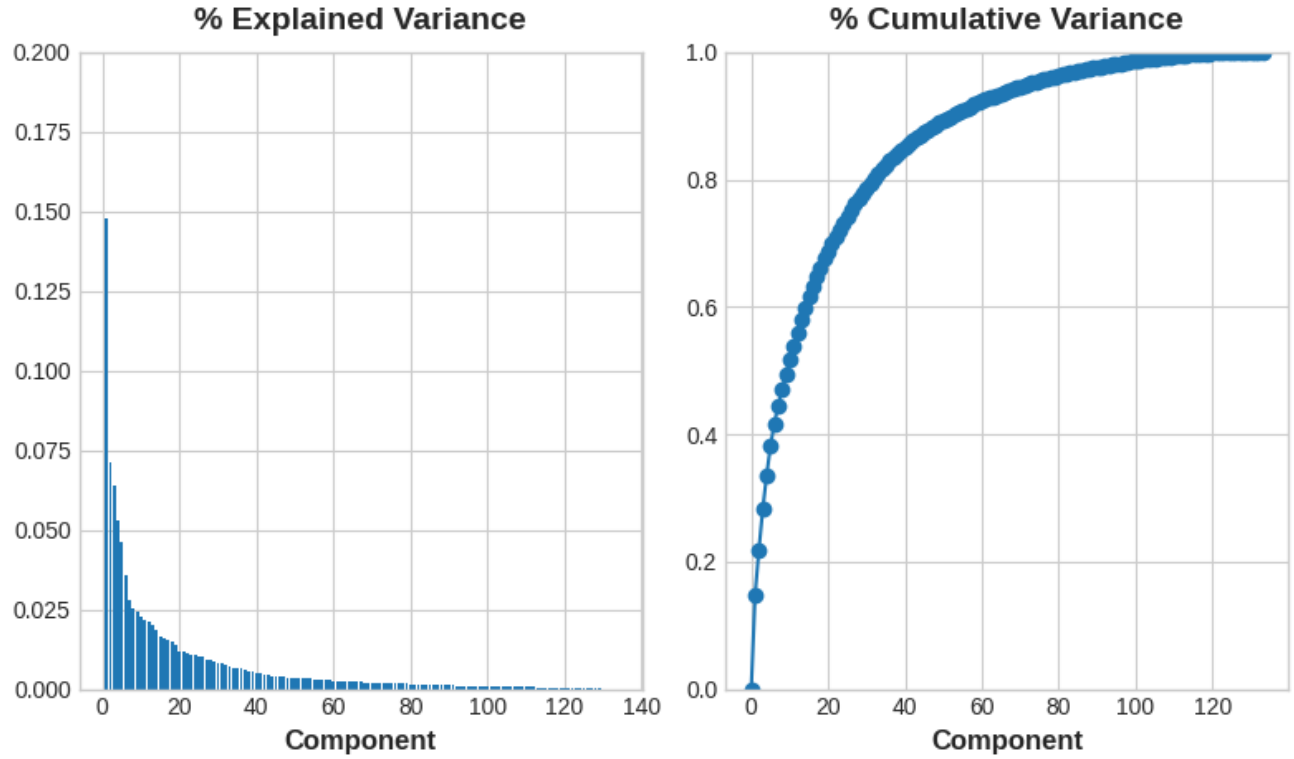
```
# Import/initialise key modules that will be used for visualising the Principal Co
from IPython.display import display

plt.style.use("seaborn-whitegrid")
plt.rc("figure", autolayout=True)
plt.rc(
    "axes",
    labelweight="bold",
    labelsize="large",
    titleweight="bold",
    titlesize=14,
    titlepad=10,
)
```

```
<ipython-input-48-d15a05cae365>:4: MatplotlibDeprecationWarning: The seaborn s
  plt.style.use("seaborn-whitegrid")
```

```
# Define a custom function that plots the explained/cumulative variances for each

def plot_variance(pca, width=8, dpi=100):
    fig, axs = plt.subplots(1, 2)
    n = pca.n_components_
    grid = np.arange(1, n + 1)

    evr = pca.explained_variance_ratio_
    axs[0].bar(grid, evr)
    axs[0].set(xlabel="Component",
               title="% Explained Variance",
               ylim=(0.0, 0.2))

    cv = np.cumsum(evr)
    axs[1].plot(np.r_[0, grid], np.r_[0, cv], "o-")
    axs[1].set(xlabel="Component",
               title="% Cumulative Variance",
               ylim=(0.0, 1.0))

    fig.set(figwidth=8, dpi=100)
    return axs


from sklearn.decomposition import PCA

# Initialise the Principal Component Analysis (PCA) algorithm.
pca = PCA()

# Fit the PCA algorithm to the validation dataset, and generate its corresponding
X_val_pca = pca.fit_transform(X_val_KMeans)

# Create a list of labels for each PC, equal in length to the number of columns ir
X_val_component_names = [f"PC{i+1}" for i in range(X_val_pca.shape[1])]

# Create a dataframe that contains the PCs generated, along with their respective
X_val_pca = pd.DataFrame(X_val_pca, columns=X_val_component_names)
```

```
# Use the custom function to plot the explained/cumulative variances, for each PC,
plot_variance(pca)
```

```
array([<Axes: title={'center': '% Explained Variance'}, xlabel='Component'>,
       <Axes: title={'center': '% Cumulative Variance'}, xlabel='Component'>],
      dtype=object)
```




```
# Calculate the cumulative sum of the explained variation ratios for the first 40
pca.explained_variance_ratio_[:40].cumsum()
```

```
array([0.14782676, 0.21895306, 0.28265223, 0.33543821, 0.38144677,
       0.41705857, 0.44500829, 0.47037174, 0.49489165, 0.51766177,
       0.53929665, 0.56022621, 0.58033055, 0.599117  , 0.61563046,
       0.63147525, 0.64703924, 0.66172336, 0.67563051, 0.68765537,
       0.69923995, 0.71044408, 0.72122103, 0.73174433, 0.74195844,
       0.75203919, 0.76123972, 0.77021924, 0.77909299, 0.78708367,
       0.79501209, 0.80263371, 0.80992618, 0.81671497, 0.82327441,
       0.82967789, 0.83570296, 0.84124797, 0.84652776, 0.85159339])
```

As shown in the cells above, the first 40 PCs contain just over 80% of the cumulative variance in the validation dataset.

This means that we can still capture a significant majority of the dataset's cumulative variance, were we to use a lower dimensionality feature-space instead, rather than simply using all features together.

```
# Create a dataframe which displays each principal component's loading/s on each o

X_val_loadings = pd.DataFrame(
    pca.components_.T,  # We need to transpose the matrix of loadings.
    columns=X_val_component_names,  # Columns are set as the principal components.
    index=X_val_KMeans.columns,  # Rows are set as the original features.
)
X_val_loadings
```

	PC1	PC2	PC3	PC4	PC5	PC6	P
Product_Info_1	-0.003794	-0.002817	-0.013725	-0.002994	0.018925	-0.012678	0.0199
Product_Info_3	0.005680	-0.010101	-0.099699	-0.031197	-0.026741	-0.037806	0.0448
Product_Info_4	-0.012015	-0.104664	-0.070933	-0.028164	0.004990	-0.096209	0.1030
Product_Info_5	0.001900	-0.002519	-0.007133	-0.000482	-0.000478	-0.004999	0.0011
Product_Info_6	-0.004604	0.003765	0.031883	0.020872	-0.035022	0.107382	-0.2224
...	...	...	...	...	...	...	
KMeansCluster_10	-0.045677	0.167227	0.000624	-0.033457	-0.110535	0.051190	0.0302
KMeansCluster_11	-0.082279	0.040796	0.161549	0.048191	0.095512	-0.084343	-0.0156
KMeansCluster_12	0.050554	-0.111340	0.185923	-0.163251	0.072364	0.059610	-0.0868
KMeansCluster_13	-0.111407	-0.048249	0.125397	0.053657	-0.049620	0.165075	0.0233
KMeansCluster_14	0.035181	-0.015094	0.027981	0.078474	0.027619	-0.066419	0.0416

133 rows × 133 columns

We can also display how each principal component is comprised, in terms of the original features' loadings.

Hence, when looking in the top 40 principal components, we can see which of the original features contribute most strongly towards the cumulative variance of the validation dataset.

```
X_val_loadings_first40 = X_val_loadings.iloc[:, :40]
X_val_loadings_first40
```

	PC1	PC2	PC3	PC4	PC5	PC6	P
Product_Info_1	-0.003794	-0.002817	-0.013725	-0.002994	0.018925	-0.012678	0.019
Product_Info_3	0.005680	-0.010101	-0.099699	-0.031197	-0.026741	-0.037806	0.044
Product_Info_4	-0.012015	-0.104664	-0.070933	-0.028164	0.004990	-0.096209	0.103
Product_Info_5	0.001900	-0.002519	-0.007133	-0.000482	-0.000478	-0.004999	0.001
Product_Info_6	-0.004604	0.003765	0.031883	0.020872	-0.035022	0.107382	-0.222
...	...	...	...	...	...	...	
KMeansCluster_10	-0.045677	0.167227	0.000624	-0.033457	-0.110535	0.051190	0.030
KMeansCluster_11	-0.082279	0.040796	0.161549	0.048191	0.095512	-0.084343	-0.015
KMeansCluster_12	0.050554	-0.111340	0.185923	-0.163251	0.072364	0.059610	-0.086
KMeansCluster_13	-0.111407	-0.048249	0.125397	0.053657	-0.049620	0.165075	0.023
KMeansCluster_14	0.035181	-0.015094	0.027981	0.078474	0.027619	-0.066419	0.041

133 rows × 40 columns

```
useful_cols = []

# Visualise which columns in the top 40 PCs contain notable variance (i.e. where t
for col in X_val_loadings_first40.columns:
    cols = X_val_loadings_first40[col].loc[abs(X_val_loadings_first40[col]) > 0.25
    cols_df = pd.DataFrame(cols)
    useful_cols.append(cols_df)

useful_cols
```

```
KMeansCluster_6  0.300434
KMeansCluster_9 -0.471953,
                        PC34
Medical_Keyword_10  0.383106
KMeansCluster_2    -0.363086
KMeansCluster_6    -0.264842
KMeansCluster_8     0.314812
KMeansCluster_9    -0.255505
KMeansCluster_13    0.250820,
                        PC35
InsuredInfo_1      -0.269289
Medical_History_9   0.532454
Medical_History_21  0.283832
Medical_Keyword_10 -0.298954
Medical_Keyword_22  0.409774,
                        PC36
InsuredInfo_1      -0.329667
Medical_History_8   0.304829
Medical_Keyword_10  0.518443,
                        PC37
InsuredInfo_1       0.360616
Medical_History_9   0.648684
Medical_Keyword_10  0.416977,
                       PC38
KMeansCluster_2 -0.460698
KMeansCluster_4 -0.261265
KMeansCluster_7  0.382604
KMeansCluster_9  0.490443,
                     PC39
Ins_Age       -0.273767
InsuredInfo_1  0.671737,
                        PC40
Medical_Keyword_1   0.587172
Medical_Keyword_30  0.423762
KMeansCluster_2     0.298809
KMeansCluster_14   -0.334293]
```

The cell above indicates which columns/features appear most commonly throughout the top 40 PCs (which contain over 80% of the cumulative variance in the dataset).

The more frequently a column name appears above, the more indicative this is that the column is useful for capturing significant variance within the dataset. It is thus more important for the models to place greater importance on these features later on.

```
Start coding or generate with AI.
```

3.3.4 Method 4: Lasso (L1) Regularisation

Finally, we will perform feature selection via L1 (lasso) regularisation using functions that are readily available in sklearn - for more information, see this link: [Feature Selection via scikit-learn]

```
# Import key modules in order to perform LASSO-based feature selection.
from sklearn.svm import LinearSVC
```

```
from sklearn.feature_selection import SelectFromModel

# Establish the Lasso (L1) Regularisation model that will perform feature selectio
linearsvc = LinearSVC(penalty="l1", dual=False, tol=1e-3, C=1e-2, random_state=0).
model = SelectFromModel(linearsvc, prefit=True)

# Reduce the dataset to the most important features, using the regularisation mode
X_val_L1 = model.transform(X_val_KMeans)

# Convert the transformed dataset into a dataframe with the same size/shape as the
# For features that were previously removed, this dataset will now include zeroes
selected_features = pd.DataFrame(model.inverse_transform(X_val_L1),
                                 index=X_val_KMeans.index,
                                 columns=X_val_KMeans.columns)

# Drop columns from the dataframe where features are deemed unimportant in capturi
# To achieve this, we selectively drop columns where their variance is equal to 0
X_val_L1reg = selected_features.drop(selected_features.columns[selected_features.v
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1250: Convergence
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:458: UserWarning: X ha
  warnings.warn(
```

```
# Determine the number of columns that are kept after L1 regularisation.

len(selected_features.columns[selected_features.var() != 0])
```

```
50
```

We have now reduced our dataset down to 57 features, from a starting value of 126.

Next, we will transform our training/test subsets so that they are now constrained to the same set of columns as derived above.

```
# Produce a separate list containing the columns preserved after L1 regularisation
selected_columns = selected_features.columns[selected_features.var() != 0]

# Reduce the training/test datasets to the same set of columns.
X_train_L1reg = X_train_KMeans[selected_columns]
X_test_L1reg = X_test_KMeans[selected_columns]
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

4. Supervised Methods

4.1 Define Models

In this project, we aim to predict the target (Response) as well as the likelihood of belonging to the predicted group using the following approaches:

Logistic/Softmax Regression

Softmax Regression is a generalised form of the standard logistic regression model. Given an instance
$\mathbf{X}$, input features are supplied to the model in order to compute a score for each class - then, the probability of $\mathbf{X}$ belonging to each class is estimated by applying the softmax function (which calculates the exponential of every score, then normalises them). Then, the algorithm returns a prediction as the class with the highest estimated probability (simply the class with the highest score).

Gaussian Naive Bayes

Gaussian Naive Bayes works as an extension of Naive Bayes (which is termed as "naive" due to its main assumption that all features are independent), by instead assuming that each feature within the data can be represented by a normal/Gaussian distribution. As a result, we only need to calculate the mean/standard deviation of each Gaussian distribution in order to predict the likelihood of an instance belonging to a given class, reducing computational complexity as compared to using other Bayesian methods.

Support Vector Machines

Support Vector Machines are capable of performing linear or nonlinear classification, and are particularly well-suited for classification of complex small- and medium-sized datasets/segmentation of high dimensionality feature spaces. They essentially work by fitting the widest possible margin between each of the classes, as a function of each of the dataset's features. As can be imagined, this technique is sensitive to feature scaling, which is why min-max scaling was performed earlier above.

Decision Trees/Random Forests

Decision Trees are powerful and versatile algorithms that are capable of fitting to/classifying complex datasets. They are generated by splitting a training dataset into two subsets recursively, based on a single feature and a corresponding threshold value (which are chosen as the pair that produces subsets with the lowest possible Gini impurity).

Random Forest Classification works by training multiple decision trees, based on the random sampling (with replacement) of a training dataset. Input features from an unseen dataset can then be supplied to each trained decision tree in order to generate a prediction, which is

subsequently averaged across all predictions to produce a final classification output; averaging across all predictions has the benefit of reducing overfitting to any given random sample within the training set.

Gradient Boosting Classifiers

Boosting refers to any ensemble method that combines several weak "learners" into a strong learner. Usually, this is accomplished by training predictors sequentially, where each tries to correct its predecessor.

With **AdaBoost classifiers**, a base classifier (such as a Decision Tree) is trained and used to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances, and then trains a second classifier using the updated weights, before making a new set of predictions. This loop is usually repeated up until there is no significant improvement in the last round's performance, or until the maximum number of iterations has been reached.

In the case of **Gradient Boosting classifiers**, the algorithm works very similarly to that used for training AdaBoost classifiers, which is by sequentially adding predictors to an ensemble - except for that, instead of updating instance weights at every iteration, the next predictor is instead fitted to the residual errors made by the previous predictor. Over the course of several iterations, this can lead to a highly refined model which is capable of recognising complex nuances in the training dataset and providing accurate classifications, however care must also be taken to avoid overfitting.

```
# Import the classification models from sklearn/xgboost.

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientB
from xgboost import XGBClassifier
```

4.2 Optimize Hyperparameters

Here, we will utilise exhaustive grid-search methods in order to optimise several of the models' hyperparameters. These are:

LogisticRegression

`tol` represents the tolerance for the stopping criteria, i.e. the minimum step size of the loss function's improvement. `C` represents the inverse of the regularisation strength, thus determining how strongly the model fits to the data.

GaussianNB

`var_smoothing` represents the degree to which we widen/"smooth" our Gaussian distributions, to account for additional samples that are further away from the distribution mean.
`SVC(kernel="linear")`

`C` represents the inverse of the regularisation strength, thus determining how strongly the model fits to the data. `tol` represents the tolerance for the stopping criteria, i.e. the minimum step size of the loss function's improvement.

LinearSVC

`tol` represents the tolerance for the stopping criteria, i.e. the minimum step size of the loss function's improvement. `C` represents the inverse of the regularisation strength, thus determining how strongly the model fits to the data. `SVC(kernel="poly")`

`C` represents the inverse of the regularisation strength, thus determining how strongly the model fits to the data. degree represents the degree of the polynomial kernel function that we want the model to fit. `tol` represents the tolerance for the stopping criteria, i.e. the minimum step size of the loss function's improvement. `SVC(kernel="rbf")`

`C` represents the inverse of the regularisation strength, thus determining how strongly the model fits to the data. `tol` represents the tolerance for the stopping criteria, i.e. the minimum step size of the loss function's improvement. `SVC(kernel="sigmoid")`

`C` represents the inverse of the regularisation strength, thus determining how strongly the model fits to the data. `tol` represents the tolerance for the stopping criteria, i.e. the minimum step size of the loss function's improvement.

DecisionTreeClassifier

`max_depth` represents the maximum depth of the tree, thus determining how strongly the model fits to the data. `max_features` represents the number of features to consider, when looking for the optimum split.

RandomForestClassifier

`n_estimators` represents the number of decision trees that are implemented by the random forest classifier. `max_depth` represents the maximum depth of the tree, thus determining how strongly the model fits to the data. `max_features` represents the number of features to consider, when looking for the optimum split.

AdaBoostClassifier

`n_estimators` represents the maximum number of boosting rounds/decision trees to be used, at which point the boosting process is terminated. `learning_rate` represents the contribution weighting applied to each decision tree, at each boosting iteration.

GradientBoostingClassifier

`learning_rate` represents the contribution weighting applied to each decision tree, at each boosting iteration. `n_estimators` represents the maximum number of boosting rounds/decision trees to be used, at which point the boosting process is terminated.

XGBClassifier

`n_estimators` represents the maximum number of boosting rounds/decision trees to be used, at which point the boosting process is terminated. learning_rate represents the contribution weighting applied to each decision tree, at each boosting iteration.

More information regarding exhaustive grid-search methods can be found at the following page: Exhaustive grid-search methods for tuning hyperparameters - scikit-learn.

```
# Import the GridSearchCV function from sklearn.

from sklearn.model_selection import GridSearchCV
```

4.2.1 Initialise each model

We set up each of the classifiers as baseline models, by initialising each model as a new object.

```
## Initialise each classifier - these are capable of providing predictions as well

### Logistic/softmax regressors
Model1_Base = LogisticRegression(random_state=0,
                                 solver='liblinear')

### Naive Bayes classifiers
Model2_Base = GaussianNB()

### Support Vector Machines (linear/non-linear)
Model3_Base = SVC(kernel='linear',
                  probability=True,
                  max_iter=1000,
                  random_state=0)

Model4_Base = LinearSVC(dual=False,
                        random_state=0,
                        max_iter=1000)

Model5_Base = SVC(kernel='poly',
                  probability=True,
                  max_iter=1000,
                  random_state=0)

Model6_Base = SVC(kernel='rbf',
                  probability=True,
                  max_iter=1000,
```

```
                  random_state=0)

Model7_Base = SVC(kernel='sigmoid',
                  probability=True,
                  max_iter=1000,
                  random_state=0)

### Decision Trees
Model8_Base = DecisionTreeClassifier(random_state=0)

### Random Forests
Model9_Base = RandomForestClassifier(n_jobs=-1,
                                     random_state=0)

### Gradient Boosting Machines
Model10_Base = AdaBoostClassifier(random_state=0)

Model11_Base = GradientBoostingClassifier(random_state=0)

Model12_Base = XGBClassifier(random_state=0,
                             n_jobs=-1,
                             eval_metric="merror")
```

4.2.2 Declare the hyperparameter grids for each model

Next, we declare the models' parameter grids that we require the grid-searches to be performed over.

```
# Config for model 1 - LogisticRegression.
param_grid_model1 = {'tol': [0.1, 0.01, 0.001, 0.0001, 0.00001, 0.000001],
                     'C': [100.0, 10.0, 1.0, 0.1, 0.01, 0.001]}


# Config for model 2 - GaussianNB.
param_grid_model2 = {'var_smoothing': [1e-01, 1e-02, 1e-03, 1e-04, 1e-05, 1e-06, 1


# Config for model 3 - SVC (linear kernel).
param_grid_model3 = {'C': [100, 10, 1.0, 0.1, 0.01, 0.001],
                     'tol': [1e-01, 1e-02, 1e-03, 1e-04, 1e-05, 1e-06]}


# Config for model 4 - LinearSVC.
param_grid_model4 = {'tol': [1e-01, 1e-02, 1e-03, 1e-04, 1e-05, 1e-06],
                     'C': [100, 10, 1.0, 0.1, 0.01, 0.001]}


# Config for model 5 - SVC (polynomial kernel).
param_grid_model5 = {'C': [100, 10, 1.0, 0.1, 0.01, 0.001],
                     'degree': [0, 1, 2, 3, 4, 5, 6],
                     'tol': [1e-01, 1e-02, 1e-03, 1e-04, 1e-05, 1e-06]}
```

```
# Config for model 6 - SVC (rbf kernel).
param_grid_model6 = {'C': [100, 10, 1.0, 0.1, 0.01, 0.001],
                     'tol': [1e-01, 1e-02, 1e-03, 1e-04, 1e-05, 1e-06]}


# Config for model 7 - SVC (sigmoid kernel).
param_grid_model7 = {'C': [100, 10, 1.0, 0.1, 0.01, 0.001],
                     'tol': [1e-01, 1e-02, 1e-03, 1e-04, 1e-05, 1e-06]}


# Config for model 8 - DecisionTreeClassifier.
param_grid_model8 = {'max_depth': [5, 10, 15, 20, 25, 30, 35, 40],
                     'max_features': [0.2, 0.4, 0.6, 0.8, 1.0]}


# Config for model 9 - RandomForestClassifier.
param_grid_model9 = {'n_estimators': [10, 100, 250, 500, 1000],
                     'max_depth': [5, 10, 15, 20, 25, 30, 35, 40],
                     'max_features': [0.2, 0.4, 0.6, 0.8, 1.0]}


# Config for model 10 - AdaBoostClassifier.
param_grid_model10 = {'n_estimators': [10, 100, 250, 500, 1000],
                      'learning_rate': [1e-0, 1e-01, 1e-02, 1e-03, 1e-04, 1e-05]}


# Config for model 11 - GradientBoostingClassifier.
param_grid_model11 = {'learning_rate': [1e-0, 1e-01, 1e-02, 1e-03, 1e-04, 1e-05],
                      'n_estimators': [10, 100, 250, 500, 1000]}


# Config for model 12 - XGBClassifier.
param_grid_model12 = {'n_estimators': [10, 100, 250, 500, 1000],
                      'learning_rate': [1e-0, 1e-01, 1e-02, 1e-03, 1e-04, 1e-05]}

# Create a list containing each of the baseline models.
BaseModels = [Model1_Base, Model2_Base, Model3_Base,
              Model4_Base, Model5_Base, Model6_Base,
              Model7_Base, Model8_Base, Model9_Base,
              Model10_Base, Model11_Base, Model12_Base]

# Create a list containing each of the model's parameter-grid dictionaries.
ParamGrids = [param_grid_model1, param_grid_model2, param_grid_model3,
              param_grid_model4, param_grid_model5, param_grid_model6,
              param_grid_model7, param_grid_model8, param_grid_model9,
              param_grid_model10, param_grid_model11, param_grid_model12]
```

4.2.3 Define a custom gridsearch function to time each experiment

In order to run each grid-search consecutively in a single loop, we define a custom function that performs a grid-search optimisation for a given model, and returns an object containing a set of optimised hyperparameters that can be easily retrieved afterwards.

```
from datetime import datetime
import pytz

# Define a custom function that performs a grid-search for a given model and paran
def GridSearcher(model, param_grid):
    ## Startup
    timezone = pytz.timezone('Europe/London')
    start_time = datetime.now(timezone)
    print("Running GridSearchCV for:", str(model))
    print("Starting at:",start_time.strftime("%H:%M:%S"))

    ## Perform the grid-search
    GridSearcher = GridSearchCV(model, param_grid, scoring='balanced_accuracy', re
    GridSearcher.fit(X_valid_L1reg, y_valid)
    # HOLDOUT n_jobs=-1

    ## Finish
    finish_time = datetime.now(timezone)
    print("Finished at:", finish_time.strftime("%H:%M:%S"))
    duration = finish_time - start_time
    dur = divmod(duration.seconds, 60)
    print("Duration: ", dur[0], 'minutes', dur[1], 'seconds')
    return GridSearcher
```

4.2.4 Run grid-searches for all models

For each model, we loop through the custom grid-search function and retrieve its optimised hyperparameters.

This section takes several hours to run.

```
##     tz = pytz.timezone('Europe/London')
##     start = datetime.now(tz)
##     print("GridSearchCV runs starting at:",start.strftime("%H:%M:%S"))

##     for i in range(0, 12):
##         GridSearchResult = GridSearcher(BaseModels[i], ParamGrids[i])
##         print(GridSearchResult.best_params_)
##         GridSearchResults.append(GridSearchResult)

##     finish = datetime.now(tz)
##     print("GridSearchCV runs finished at:", finish.strftime("%H:%M:%S"))
##     duration = finish - start
##     dur = divmod(duration.seconds, 60)
##     print("Total Duration: ", dur[0], 'minutes', dur[1], 'seconds')
```

```
##     for i in GridSearchResults:
##         print(i.best_params_)


# The following sets of optimised hyperparameters were obtained via the cell above

Model1_Opt_Params = {'C': 100.0, 'tol': 0.01}
Model2_Opt_Params = {'var_smoothing': 0.0001}
Model3_Opt_Params = {'C': 1.0, 'tol': 0.1}
Model4_Opt_Params = {'C': 100, 'tol': 0.001}
Model5_Opt_Params = {'C': 0.1, 'degree': 6, 'tol': 0.1}
Model6_Opt_Params = {'C': 10, 'tol': 0.01}
Model7_Opt_Params = {'C': 1.0, 'tol': 0.01}
Model8_Opt_Params = {'max_depth': 10, 'max_features': 0.8}
Model9_Opt_Params = {'max_depth': 40, 'max_features': 0.6, 'n_estimators': 1000}
Model10_Opt_Params = {'learning_rate': 1.0, 'n_estimators': 100}
Model11_Opt_Params = {'learning_rate': 0.1, 'n_estimators': 250}
Model12_Opt_Params = {'learning_rate': 1.0, 'n_estimators': 10}

# Create a list containing each of the models' optimised hyperparameters.
OptimisedParams = [Model1_Opt_Params, Model2_Opt_Params, Model3_Opt_Params,
                   Model4_Opt_Params, Model5_Opt_Params, Model6_Opt_Params,
                   Model7_Opt_Params, Model8_Opt_Params, Model9_Opt_Params,
                   Model10_Opt_Params, Model11_Opt_Params, Model12_Opt_Params]
```

4.2.5 Update all baseline models to use optimised hyperparameters

Finally, we update the baseline models declared earlier above, by passing the optimised hyperparameters to each model using the `.set_params(**kwargs)` method.

```
OptimisedModels = []

# Update each of the baseline models to include their (respective) optimised hyper
for i in range(0, 12):
    OptimisedModel = BaseModels[i].set_params(**OptimisedParams[i])
    OptimisedModels.append(OptimisedModel)

OptimisedModels
```

```
[LogisticRegression(C=100.0, random_state=0, solver='liblinear', tol=0.01),
 GaussianNB(var_smoothing=0.0001),
 SVC(kernel='linear', max_iter=1000, probability=True, random_state=0,
tol=0.1),
 LinearSVC(C=100, dual=False, random_state=0, tol=0.001),
 SVC(C=0.1, degree=6, kernel='poly', max_iter=1000, probability=True,
     random_state=0, tol=0.1),
 SVC(C=10, max_iter=1000, probability=True, random_state=0, tol=0.01),
 SVC(kernel='sigmoid', max_iter=1000, probability=True, random_state=0,
tol=0.01),
 DecisionTreeClassifier(max_depth=10, max_features=0.8, random_state=0),
 RandomForestClassifier(max_depth=40, max_features=0.6, n_estimators=1000,
                        n_jobs=-1, random_state=0),
 AdaBoostClassifier(n_estimators=100, random_state=0),
```

```
 GradientBoostingClassifier(n_estimators=250, random_state=0),
 XGBClassifier(base_score=None, booster=None, callbacks=None,
               colsample_bylevel=None, colsample_bynode=None,
               colsample_bytree=None, device=None,
early_stopping_rounds=None,
               enable_categorical=False, eval_metric='merror',
               feature_types=None, gamma=None, grow_policy=None,
               importance_type=None, interaction_constraints=None,
               learning_rate=1.0, max_bin=None, max_cat_threshold=None,
               max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
               max_leaves=None, min_child_weight=None, missing=nan,
               monotone_constraints=None, multi_strategy=None,
n_estimators=10,
               n_jobs=-1, num_parallel_tree=None, random_state=0, ...)]
```

4.3 Train Models (using training data)

In this section, we will now train our optimised models on the encoded training dataset.

```
# Create a duplicate copy of the OptimisedModels list, for use in model training.
TrainedModels = OptimisedModels
```

XGBoost's classifiers require labels to be supplied in a zero-based fashion (i.e. classes 1-8 must instead be converted into labels 0-7 prior to training/prediction). Previously, this would have been handled automatically via the use of the `use_label_encoder` keyword argument - however, we must now explicitly label-encode our dataset prior to fitting/predicting as this has since become deprecated.

Hence, we will establish a label encoder that converts the classes into compatible labels.

```
# Label-encode each dataset for compatibility with the XGBoost classifier (model 1
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

# Convert each set of class labels (1-8) into encoded labels (0-7).
y_train = le.fit_transform(y_train)
y_val = le.transform(y_val)
y_test = le.transform(y_test)


#print(y_train)
#print(y_val.unique())
#print(y_test.unique())
```

We can now train all of our models using the encoded/transformed labels.

```
# Train each of the optimised models, using the training dataset.
for model in TrainedModels:
```

```
tz = pytz.timezone('Europe/London')
start = datetime.now(tz)
print("Model training started at:",start.strftime("%H:%M:%S"))
model.fit(X_train_L1reg, y_train)
finish = datetime.now(tz)
print("Model training finished at:",finish.strftime("%H:%M:%S"))
duration = finish - start
dur = divmod(duration.seconds, 60)
print("Total Duration for model training: ", dur[0], 'minutes', dur[1], 'secor
print("Model "+str(TrainedModels.index(model)+1)+" - "+model.__class__.__name_
```

```
Model training started at: 02:20:32
Model training finished at: 02:20:36
Total Duration for model training:  0 minutes 3 seconds
Model 1 - LogisticRegression has been trained.
Model training started at: 02:20:36
Model training finished at: 02:20:36
Total Duration for model training:  0 minutes 0 seconds
Model 2 - GaussianNB has been trained.
Model training started at: 02:20:36
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:297: ConvergenceW
  warnings.warn(
Model training finished at: 02:23:18
Total Duration for model training:  2 minutes 42 seconds
Model 3 - SVC has been trained.
Model training started at: 02:23:18
Model training finished at: 02:23:25
Total Duration for model training:  0 minutes 6 seconds
Model 4 - LinearSVC has been trained.
Model training started at: 02:23:25
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:297: ConvergenceW
  warnings.warn(
Model training finished at: 02:26:53
Total Duration for model training:  3 minutes 27 seconds
Model 5 - SVC has been trained.
Model training started at: 02:26:53
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:297: ConvergenceW
  warnings.warn(
Model training finished at: 02:31:06
Total Duration for model training:  4 minutes 13 seconds
Model 6 - SVC has been trained.
Model training started at: 02:31:06
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:297: ConvergenceW
  warnings.warn(
Model training finished at: 02:36:47
Total Duration for model training:  5 minutes 41 seconds
Model 7 - SVC has been trained.
Model training started at: 02:36:47
Model training finished at: 02:36:48
Total Duration for model training:  0 minutes 0 seconds
Model 8 - DecisionTreeClassifier has been trained.
Model training started at: 02:36:48
Model training finished at: 02:39:52
Total Duration for model training:  3 minutes 4 seconds
Model 9 - RandomForestClassifier has been trained.
Model training started at: 02:39:52
Model training finished at: 02:39:56
Total Duration for model training:  0 minutes 4 seconds
```

```
Model 10 - AdaBoostClassifier has been trained.
Model training started at: 02:39:56
Model training finished at: 02:42:25
Total Duration for model training:  2 minutes 28 seconds
Model 11 - GradientBoostingClassifier has been trained.
Model training started at: 02:42:25
Model training finished at: 02:42:27
Total Duration for model training:  0 minutes 2 seconds
Model 12 - XGBClassifier has been trained.
```

4.2 Calibrate Probabilities

Our models are now capable of providing predictions as well as some initial probability estimates - however, we wish to obtain true probabilities as currently, by default, ours do not yet factor in the expected distributions of the predicted classes. To this end, we will need to perform probability calibration for all of our models, so that they are capable of providing true likelihoods of belonging to each respective class.

More information on this topic can be found at the following page: [Probability Calibration - scikit-learn](scikit-learn).

```
from sklearn.calibration import CalibratedClassifierCV


# Calibrate each classifier using the validation dataset.

Model1_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[0], method="isc
Model2_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[1], method="isc
Model3_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[2], method="isc
Model4_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[3], method="isc
Model5_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[4], method="isc
Model6_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[5], method="isc
Model7_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[6], method="isc
Model8_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[7], method="isc
Model9_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[8], method="isc
Model10_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[9], method="is
Model11_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[10], method="i
Model12_Calibrated = CalibratedClassifierCV(estimator=TrainedModels[11], method="i


# Use each model to generate predictions (as probability estimates of belonging to

Model1_Valid_PredProb = Model1_Calibrated.predict_proba(X_val_L1reg)
Model2_Valid_PredProb = Model2_Calibrated.predict_proba(X_val_L1reg)
Model3_Valid_PredProb = Model3_Calibrated.predict_proba(X_val_L1reg)
Model4_Valid_PredProb = Model4_Calibrated.predict_proba(X_val_L1reg)
Model5_Valid_PredProb = Model5_Calibrated.predict_proba(X_val_L1reg)
Model6_Valid_PredProb = Model6_Calibrated.predict_proba(X_val_L1reg)
Model7_Valid_PredProb = Model7_Calibrated.predict_proba(X_val_L1reg)
Model8_Valid_PredProb = Model8_Calibrated.predict_proba(X_val_L1reg)
Model9_Valid_PredProb = Model9_Calibrated.predict_proba(X_val_L1reg)
```

```
Model10_Valid_PredProb = Model10_Calibrated.predict_proba(X_val_L1reg)
Model11_Valid_PredProb = Model11_Calibrated.predict_proba(X_val_L1reg)
Model12_Valid_PredProb = Model12_Calibrated.predict_proba(X_val_L1reg)


## Convert the probability estimates into label-based predictions, then label-enc

# Model 1 - converting probabilities into label predictions
Model1_Valid_Preds = np.argmax(Model1_Valid_PredProb, axis=1)
Model1_Valid_Classes = Model1_Calibrated.classes_
Model1_Valid_Preds = [Model1_Valid_Classes[i] for i in Model1_Valid_Preds]
Model1_Valid_Preds = le.inverse_transform(Model1_Valid_Preds)

# Model 2 - converting probabilities into label predictions
Model2_Valid_Preds = np.argmax(Model2_Valid_PredProb, axis=1)
Model2_Valid_Classes = Model2_Calibrated.classes_
Model2_Valid_Preds = [Model2_Valid_Classes[i] for i in Model2_Valid_Preds]
Model2_Valid_Preds = le.inverse_transform(Model2_Valid_Preds)

# Model 3 - converting probabilities into label predictions
Model3_Valid_Preds = np.argmax(Model3_Valid_PredProb, axis=1)
Model3_Valid_Classes = Model3_Calibrated.classes_
Model3_Valid_Preds = [Model3_Valid_Classes[i] for i in Model3_Valid_Preds]
Model3_Valid_Preds = le.inverse_transform(Model3_Valid_Preds)

# Model 4 - converting probabilities into label predictions
Model4_Valid_Preds = np.argmax(Model4_Valid_PredProb, axis=1)
Model4_Valid_Classes = Model4_Calibrated.classes_
Model4_Valid_Preds = [Model4_Valid_Classes[i] for i in Model4_Valid_Preds]
Model4_Valid_Preds = le.inverse_transform(Model4_Valid_Preds)

# Model 5 - converting probabilities into label predictions
Model5_Valid_Preds = np.argmax(Model5_Valid_PredProb, axis=1)
Model5_Valid_Classes = Model5_Calibrated.classes_
Model5_Valid_Preds = [Model5_Valid_Classes[i] for i in Model5_Valid_Preds]
Model5_Valid_Preds = le.inverse_transform(Model5_Valid_Preds)

# Model 6 - converting probabilities into label predictions
Model6_Valid_Preds = np.argmax(Model6_Valid_PredProb, axis=1)
Model6_Valid_Classes = Model6_Calibrated.classes_
Model6_Valid_Preds = [Model6_Valid_Classes[i] for i in Model6_Valid_Preds]
Model6_Valid_Preds = le.inverse_transform(Model6_Valid_Preds)

# Model 7 - converting probabilities into label predictions
Model7_Valid_Preds = np.argmax(Model7_Valid_PredProb, axis=1)
Model7_Valid_Classes = Model7_Calibrated.classes_
Model7_Valid_Preds = [Model7_Valid_Classes[i] for i in Model7_Valid_Preds]
Model7_Valid_Preds = le.inverse_transform(Model7_Valid_Preds)

# Model 8 - converting probabilities into label predictions
Model8_Valid_Preds = np.argmax(Model8_Valid_PredProb, axis=1)
Model8_Valid_Classes = Model8_Calibrated.classes_
Model8_Valid_Preds = [Model8_Valid_Classes[i] for i in Model8_Valid_Preds]
Model8_Valid_Preds = le.inverse_transform(Model8_Valid_Preds)
```

```
# Model 9 - converting probabilities into label predictions
Model9_Valid_Preds = np.argmax(Model9_Valid_PredProb, axis=1)
Model9_Valid_Classes = Model9_Calibrated.classes_
Model9_Valid_Preds = [Model9_Valid_Classes[i] for i in Model9_Valid_Preds]
Model9_Valid_Preds = le.inverse_transform(Model9_Valid_Preds)

# Model 10 - converting probabilities into label predictions
Model10_Valid_Preds = np.argmax(Model10_Valid_PredProb, axis=1)
Model10_Valid_Classes = Model10_Calibrated.classes_
Model10_Valid_Preds = [Model10_Valid_Classes[i] for i in Model10_Valid_Preds]
Model10_Valid_Preds = le.inverse_transform(Model10_Valid_Preds)

# Model 11 - converting probabilities into label predictions
Model11_Valid_Preds = np.argmax(Model11_Valid_PredProb, axis=1)
Model11_Valid_Classes = Model11_Calibrated.classes_
Model11_Valid_Preds = [Model11_Valid_Classes[i] for i in Model11_Valid_Preds]
Model11_Valid_Preds = le.inverse_transform(Model11_Valid_Preds)

# Model 12 - converting probabilities into label predictions
Model12_Valid_Preds = np.argmax(Model12_Valid_PredProb, axis=1)
Model12_Valid_Classes = Model12_Calibrated.classes_
Model12_Valid_Preds = [Model12_Valid_Classes[i] for i in Model12_Valid_Preds]
Model12_Valid_Preds = le.inverse_transform(Model12_Valid_Preds)
```

5. Performance Results

5.1 Evaluate Model Performances (using validation data)

In a real-world situation, other factors - such as training time, model explainability and costs/difficulties associated with model deployment/maintenance - will also have an impact on which model should be chosen for live use.

However, for the purposes of this project we are simply aiming to compare each of the machine learning algorithms used thus far, solely based in terms of their predictive performances, in order to select the "best" model. In the code below, a number of techniques are demonstrated in order to display how each model has performed in terms of classifying applicants into each Response group.

We will use a combination of functions across both sklearn as well as scikitplot in order to visualise each model's performance.

```
# Downgrade SciPy to version 1.11.4
!pip install scipy==1.11.4
!pip install mljar-scikit-plot
```

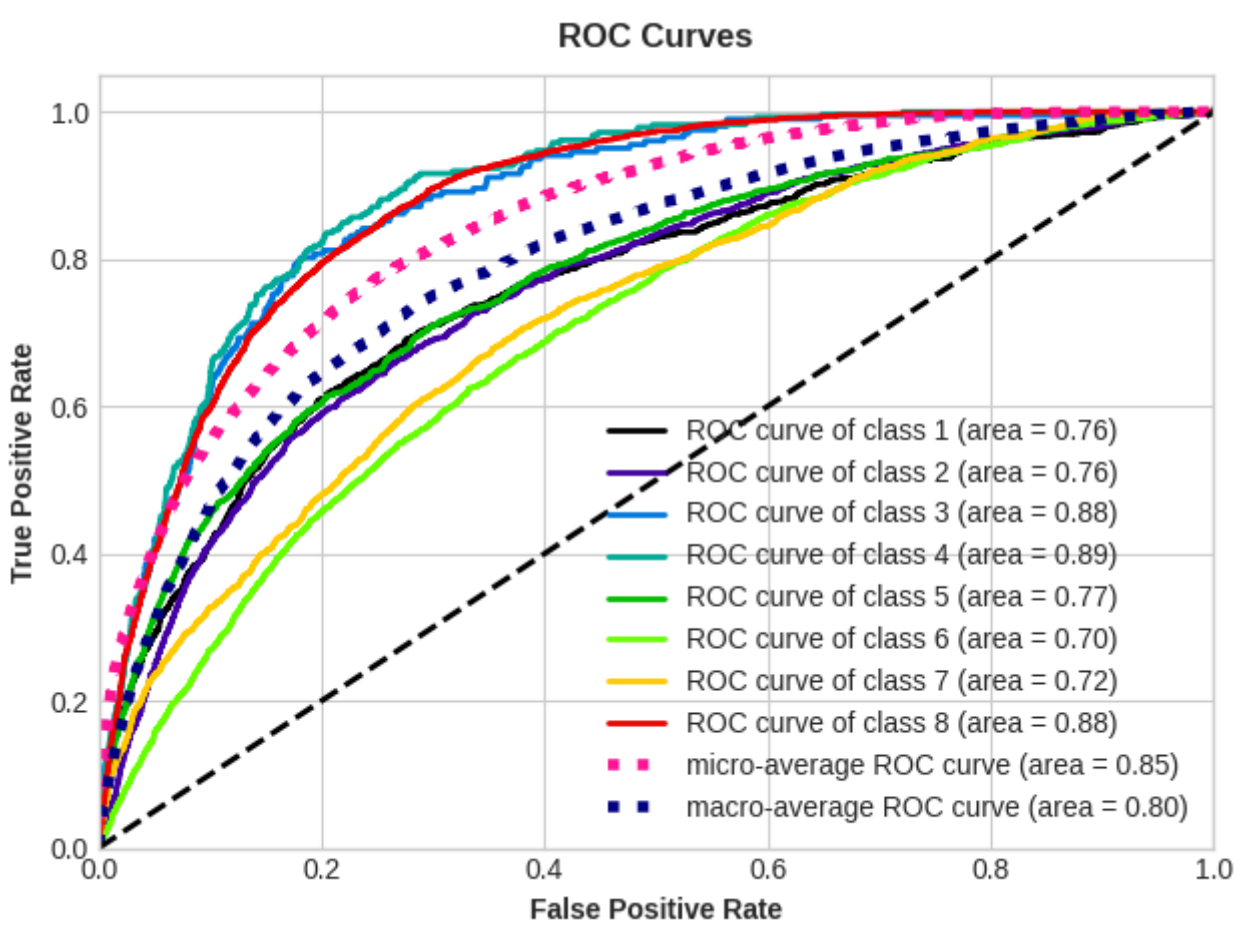
```
Requirement already satisfied: scipy==1.11.4 in /usr/local/lib/python3.10/dist
Requirement already satisfied: numpy<1.28.0,>=1.21.6 in /usr/local/lib/python3
Requirement already satisfied: mljar-scikit-plot in /usr/local/lib/python3.10/
Requirement already satisfied: matplotlib>=1.4.0 in /usr/local/lib/python3.10/
Requirement already satisfied: scikit-learn>=1.1.0 in /usr/local/lib/python3.1
Requirement already satisfied: joblib>=0.10 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-pack
```

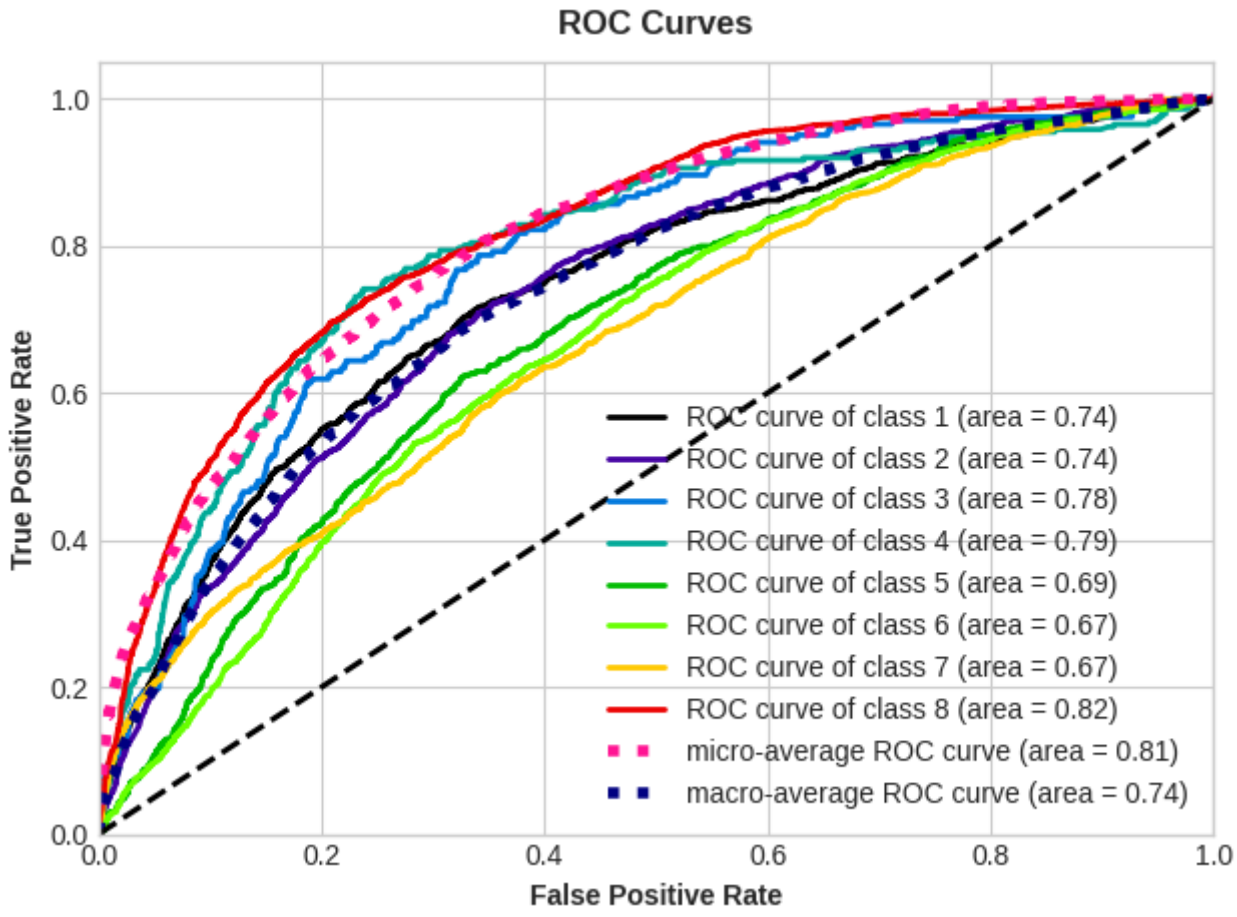
```
#!pip install --upgrade scipy
#!pip install --upgrade scikit-plot
#!pip uninstall xgboost -y
#!pip install xgboost  # Installs the CPU version by default

# Import key modules/functions for evaluating model performance.
from scikitplot.metrics import plot_roc, plot_confusion_matrix
from sklearn.metrics import classification_report, balanced_accuracy_score

# If you still need to use the 'interp' function directly, import it like this:
#from scipy.interpolate import interp


# Convert the validation dataset's encoded labels (0-7) back to the original set c
y_val = le.inverse_transform(y_val)

# Create a list containing each model's predicted labels.
Valid_Preds = [Model1_Valid_Preds, Model2_Valid_Preds, Model3_Valid_Preds,
               Model4_Valid_Preds, Model5_Valid_Preds, Model6_Valid_Preds,
               Model7_Valid_Preds, Model8_Valid_Preds, Model9_Valid_Preds,
               Model10_Valid_Preds, Model11_Valid_Preds, Model12_Valid_Preds]

# Create a list containing each model's prediction probabilities.
Valid_PredProbs = [Model1_Valid_PredProb, Model2_Valid_PredProb, Model3_Valid_Prec
                   Model4_Valid_PredProb, Model5_Valid_PredProb, Model6_Valid_Prec
                   Model7_Valid_PredProb, Model8_Valid_PredProb, Model9_Valid_Prec
                   Model10_Valid_PredProb, Model11_Valid_PredProb, Model12_Valid_P
```

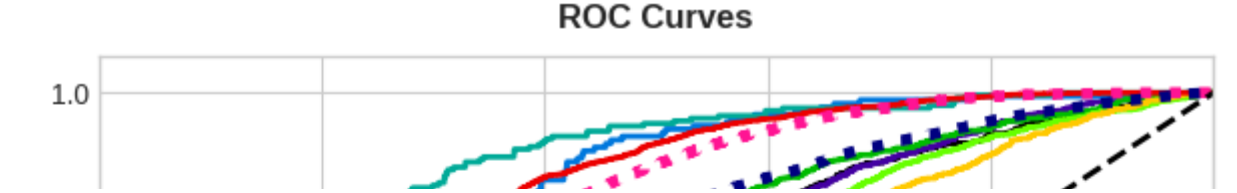
5.1.1 ROC Curves

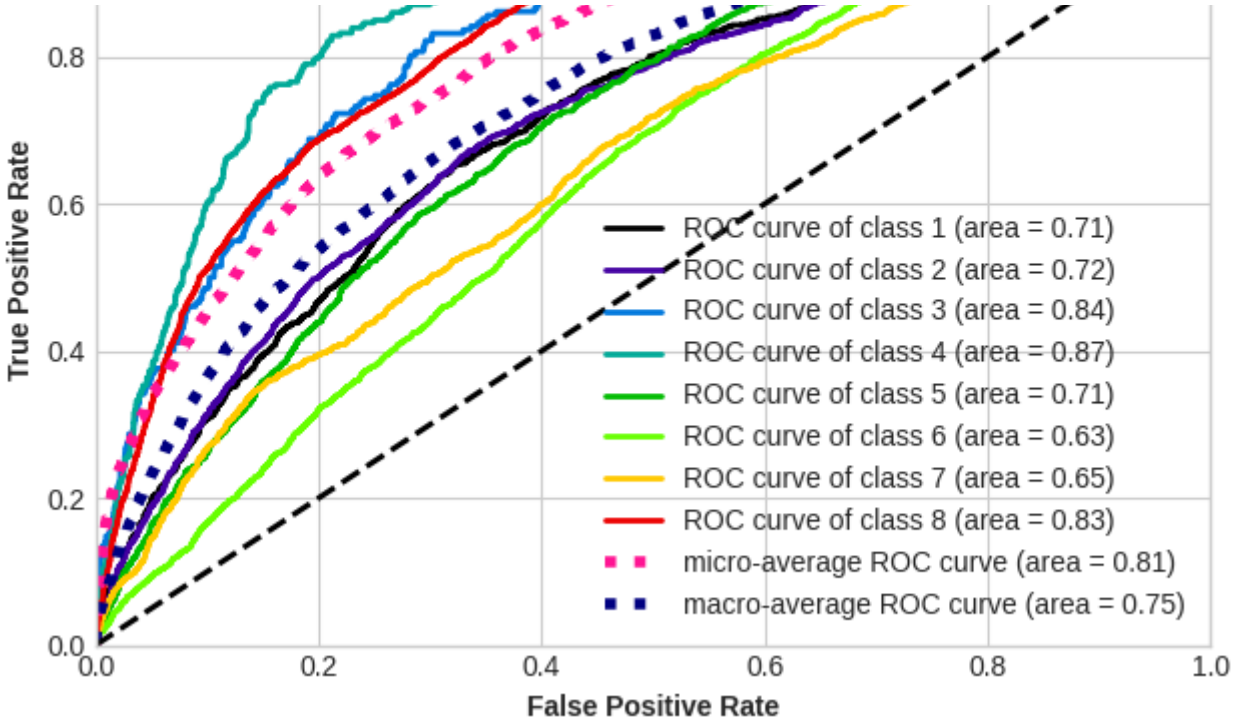
The plots below display the ROC Curves (i.e. TP rate vs. FP rate) for each of the 12 models.

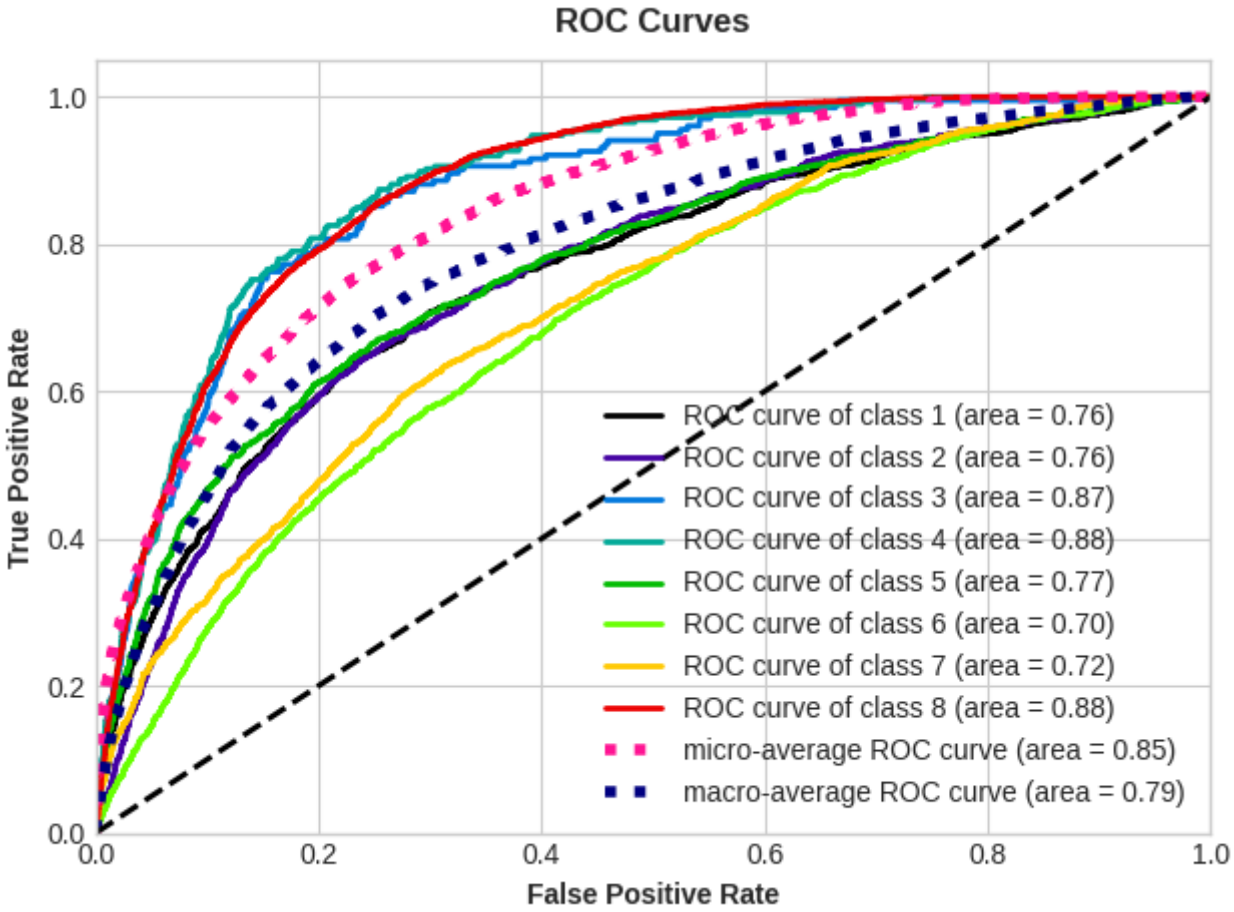
```
from scikitplot.metrics import plot_roc, plot_confusion_matrix

# For each model, plot an Receiver Operating Characteristic (ROC) curve to displa

for y_val_predprobs in Valid_PredProbs:
    plot_roc(y_val, y_val_predprobs)
```

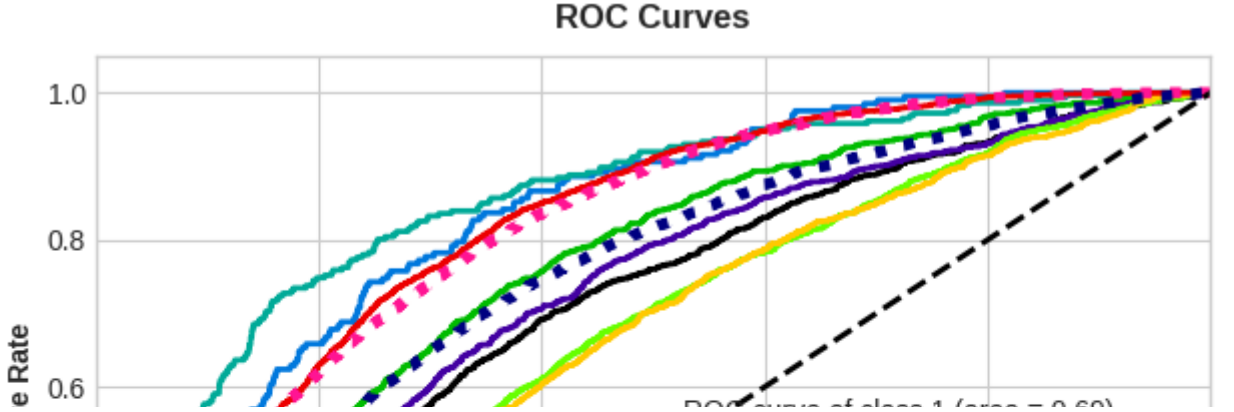

ROC Curves
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 1 (area = 0.76)
ROC curve of class 2 (area = 0.76)
ROC curve of class 3 (area = 0.88)
ROC curve of class 4 (area = 0.89)
ROC curve of class 5 (area = 0.77)
ROC curve of class 6 (area = 0.70)
ROC curve of class 7 (area = 0.72)
ROC curve of class 8 (area = 0.88)
micro-average ROC curve (area = 0.85)
macro-average ROC curve (area = 0.80)

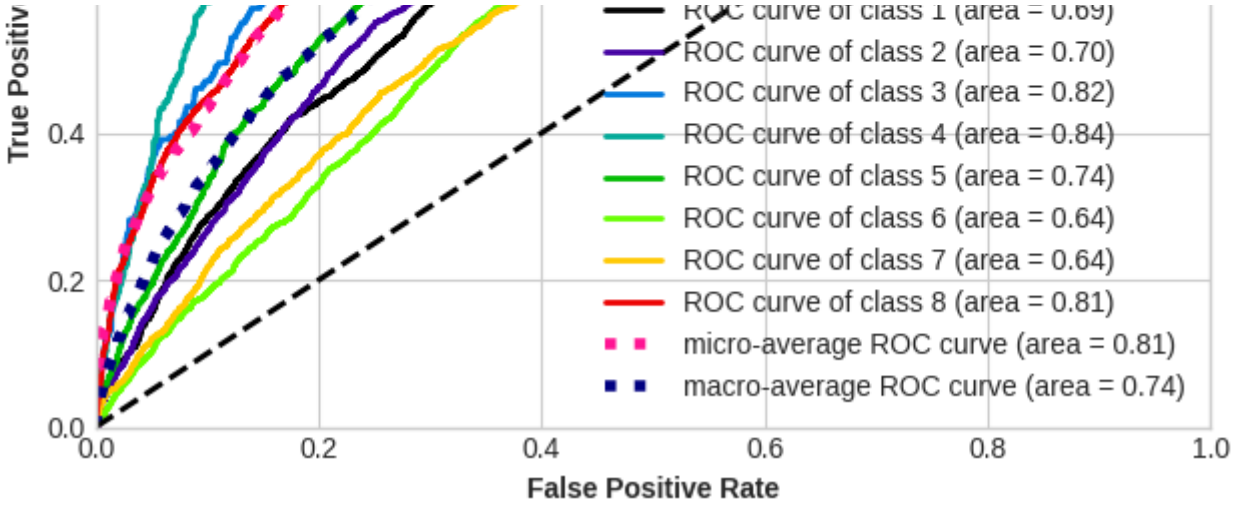

ROC Curves
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve of class 1 (area = 0.74)
ROC curve of class 2 (area = 0.74)
ROC curve of class 3 (area = 0.78)
ROC curve of class 4 (area = 0.79)
ROC curve of class 5 (area = 0.69)
ROC curve of class 6 (area = 0.67)
ROC curve of class 7 (area = 0.67)
ROC curve of class 8 (area = 0.82)
micro-average ROC curve (area = 0.81)
macro-average ROC curve (area = 0.74)

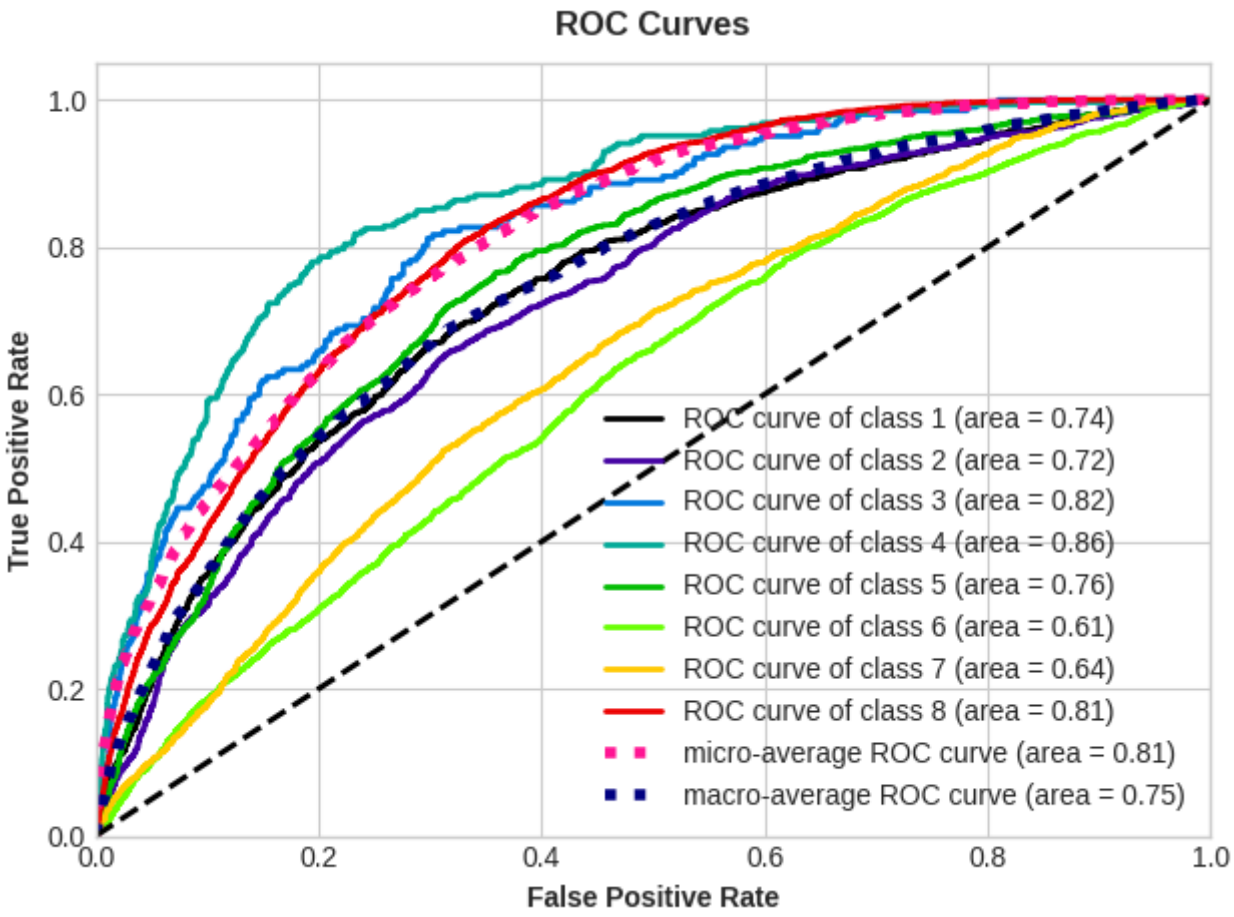

ROC Curves
1.0

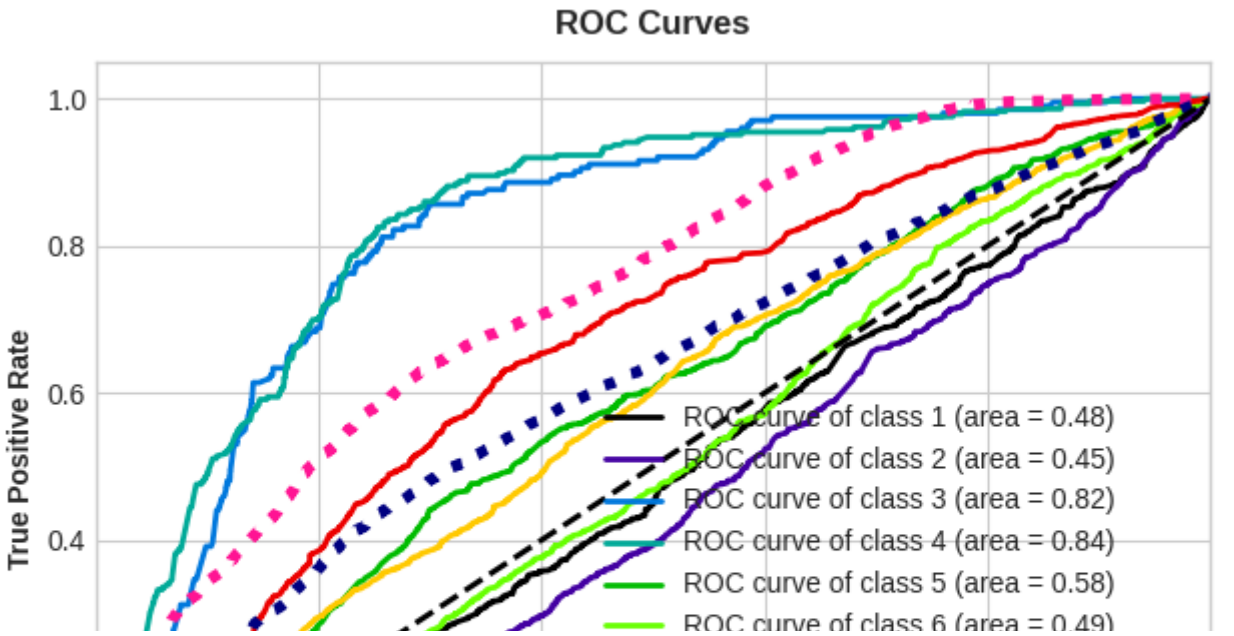

True Positive Rate
False Positive Rate
ROC curve of class 1 (area = 0.71)
ROC curve of class 2 (area = 0.72)
ROC curve of class 3 (area = 0.84)
ROC curve of class 4 (area = 0.87)
ROC curve of class 5 (area = 0.71)
ROC curve of class 6 (area = 0.63)
ROC curve of class 7 (area = 0.65)
ROC curve of class 8 (area = 0.83)
micro-average ROC curve (area = 0.81)
macro-average ROC curve (area = 0.75)

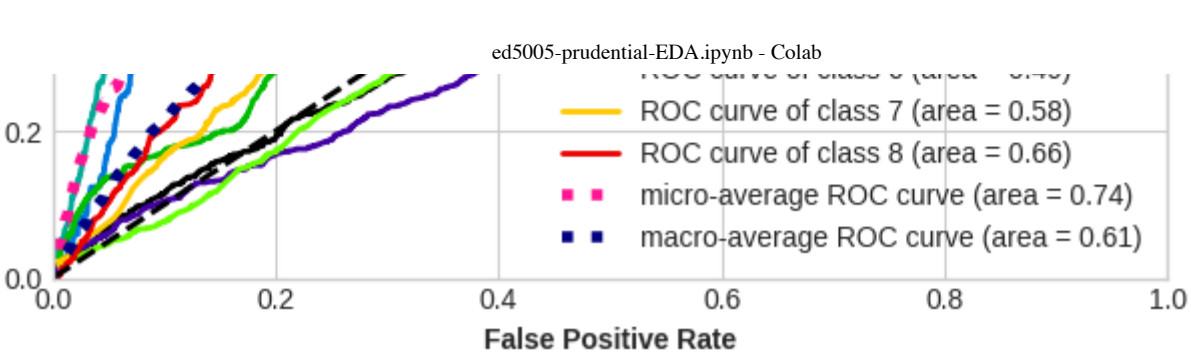

ROC Curves
True Positive Rate
False Positive Rate
ROC curve of class 1 (area = 0.76)
ROC curve of class 2 (area = 0.76)
ROC curve of class 3 (area = 0.87)
ROC curve of class 4 (area = 0.88)
ROC curve of class 5 (area = 0.77)
ROC curve of class 6 (area = 0.70)
ROC curve of class 7 (area = 0.72)
ROC curve of class 8 (area = 0.88)
micro-average ROC curve (area = 0.85)
macro-average ROC curve (area = 0.79)

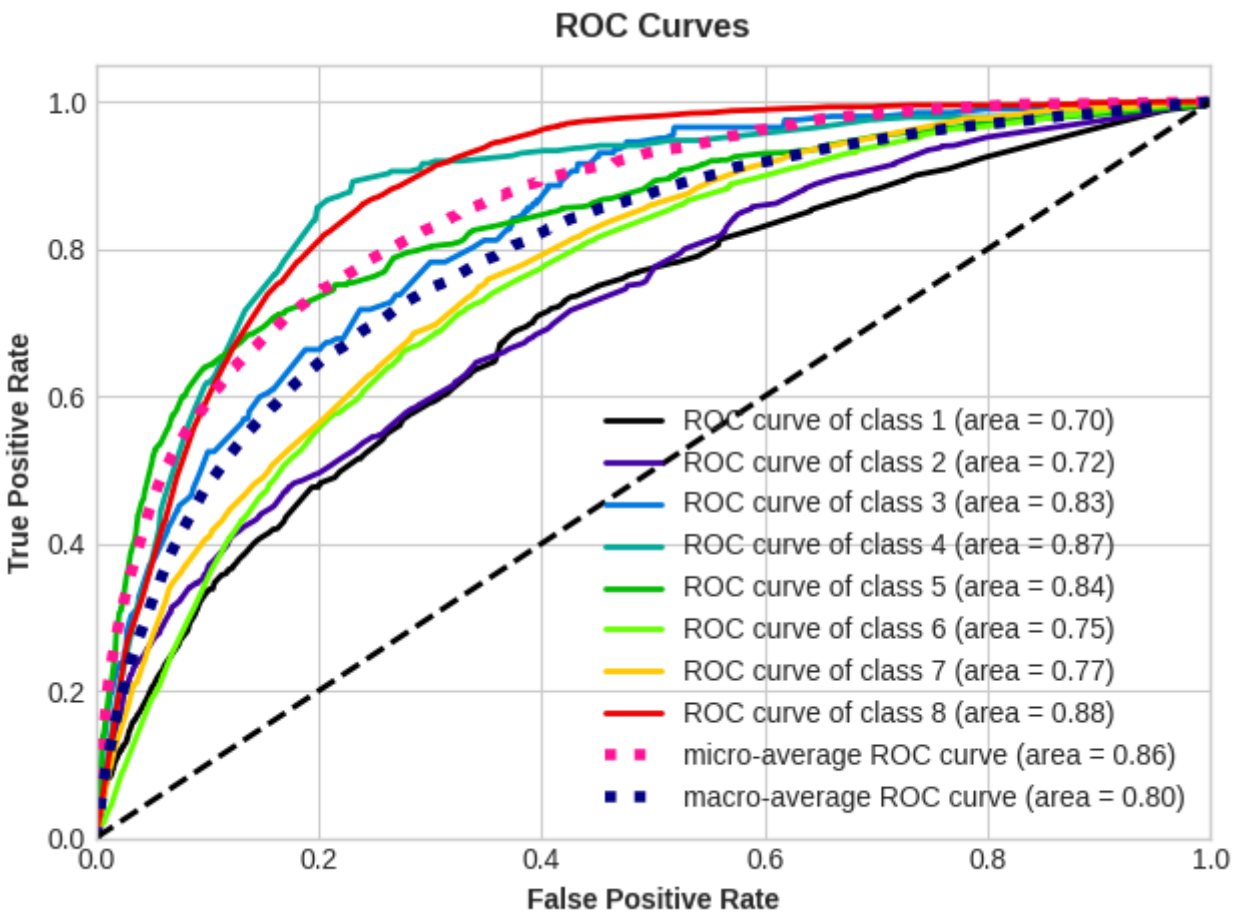

ROC Curves

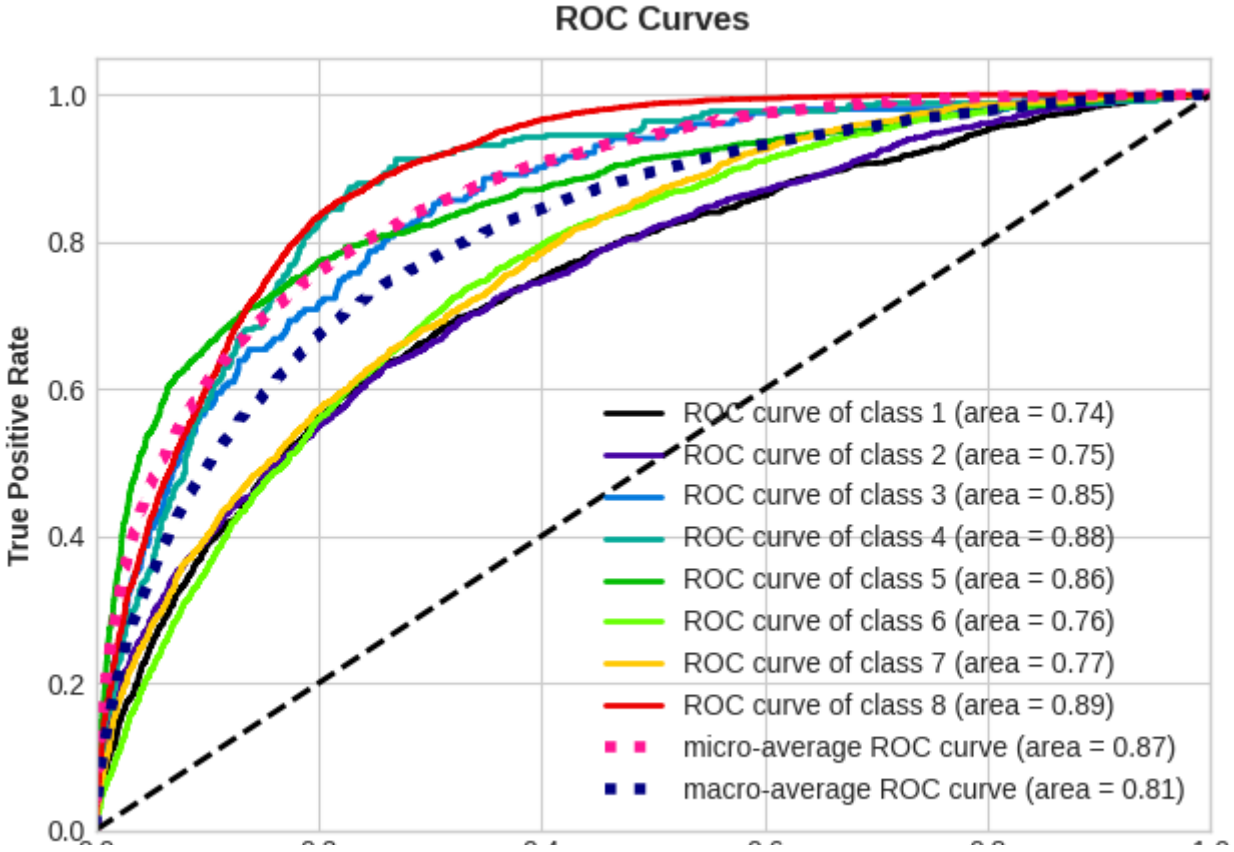

True Positiv
0.0
0.2
0.4
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
ROC curve of class 1 (area = 0.69)
ROC curve of class 2 (area = 0.70)
ROC curve of class 3 (area = 0.82)
ROC curve of class 4 (area = 0.84)
ROC curve of class 5 (area = 0.74)
ROC curve of class 6 (area = 0.64)
ROC curve of class 7 (area = 0.64)
ROC curve of class 8 (area = 0.81)
micro-average ROC curve (area = 0.81)
macro-average ROC curve (area = 0.74)

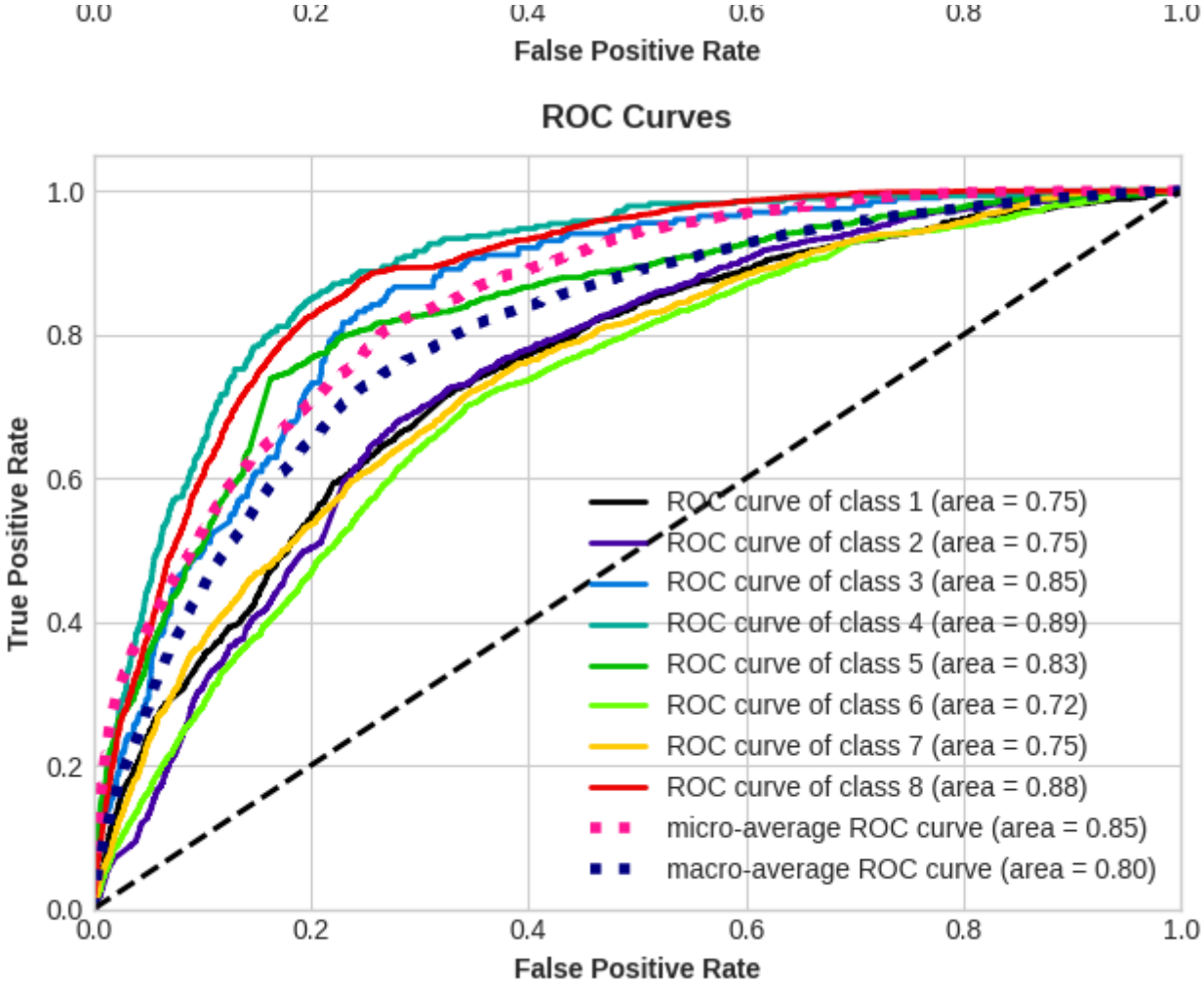

ROC Curves
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
ROC curve of class 1 (area = 0.74)
ROC curve of class 2 (area = 0.72)
ROC curve of class 3 (area = 0.82)
ROC curve of class 4 (area = 0.86)
ROC curve of class 5 (area = 0.76)
ROC curve of class 6 (area = 0.61)
ROC curve of class 7 (area = 0.64)
ROC curve of class 8 (area = 0.81)
micro-average ROC curve (area = 0.81)
macro-average ROC curve (area = 0.75)

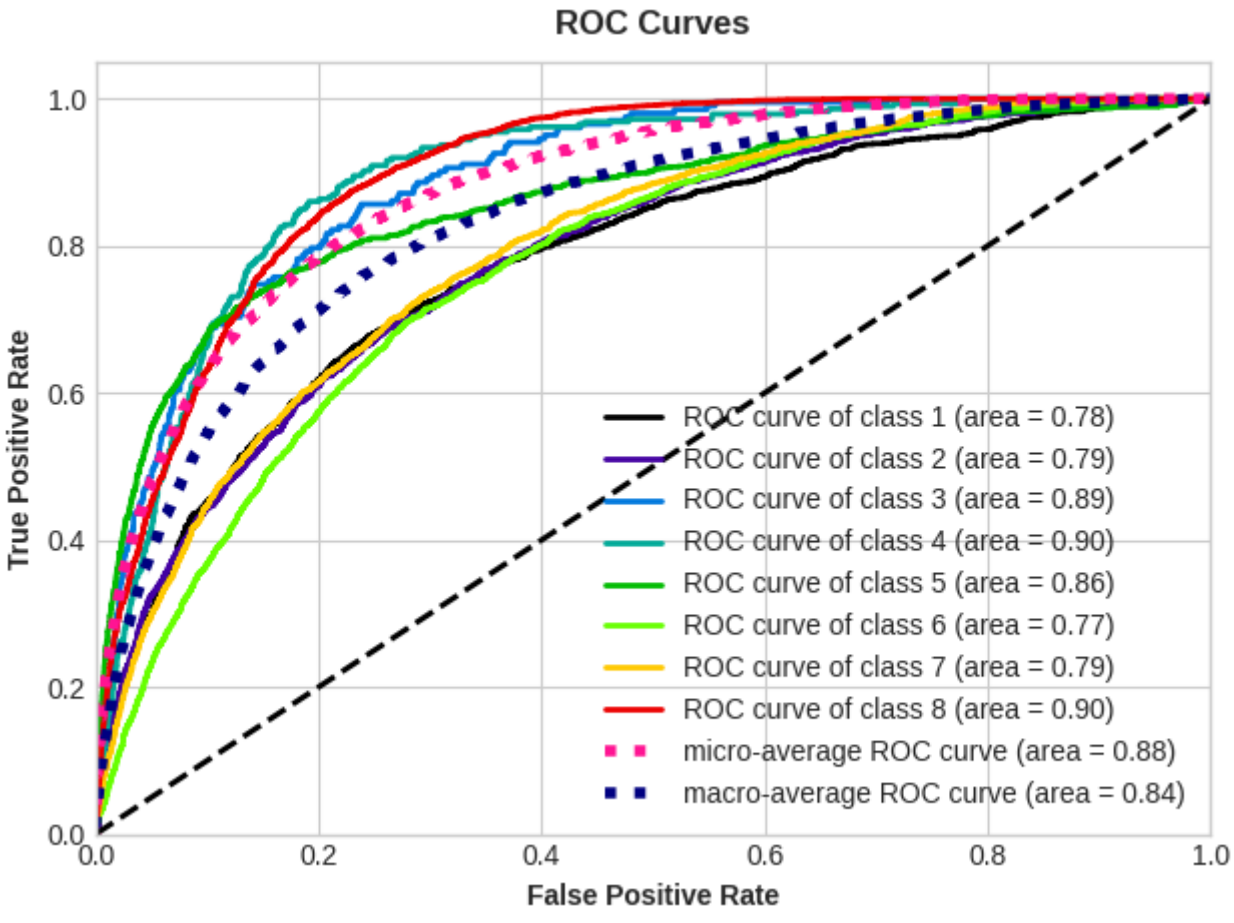

ROC Curves
True Positive Rate
0.4
0.6
0.8
1.0
ROC curve of class 1 (area = 0.48)
ROC curve of class 2 (area = 0.45)
ROC curve of class 3 (area = 0.82)
ROC curve of class 4 (area = 0.84)
ROC curve of class 5 (area = 0.58)

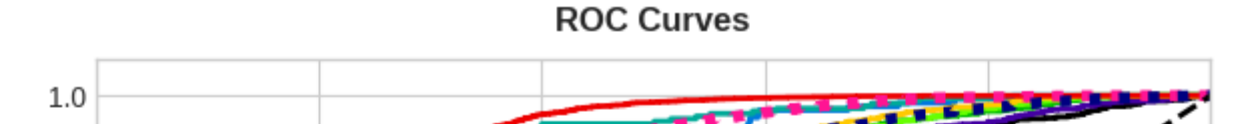

ROC curve of class 7 (area = 0.58)
ROC curve of class 8 (area = 0.66)
micro-average ROC curve (area = 0.74)
macro-average ROC curve (area = 0.61)
0.0
0.2
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate

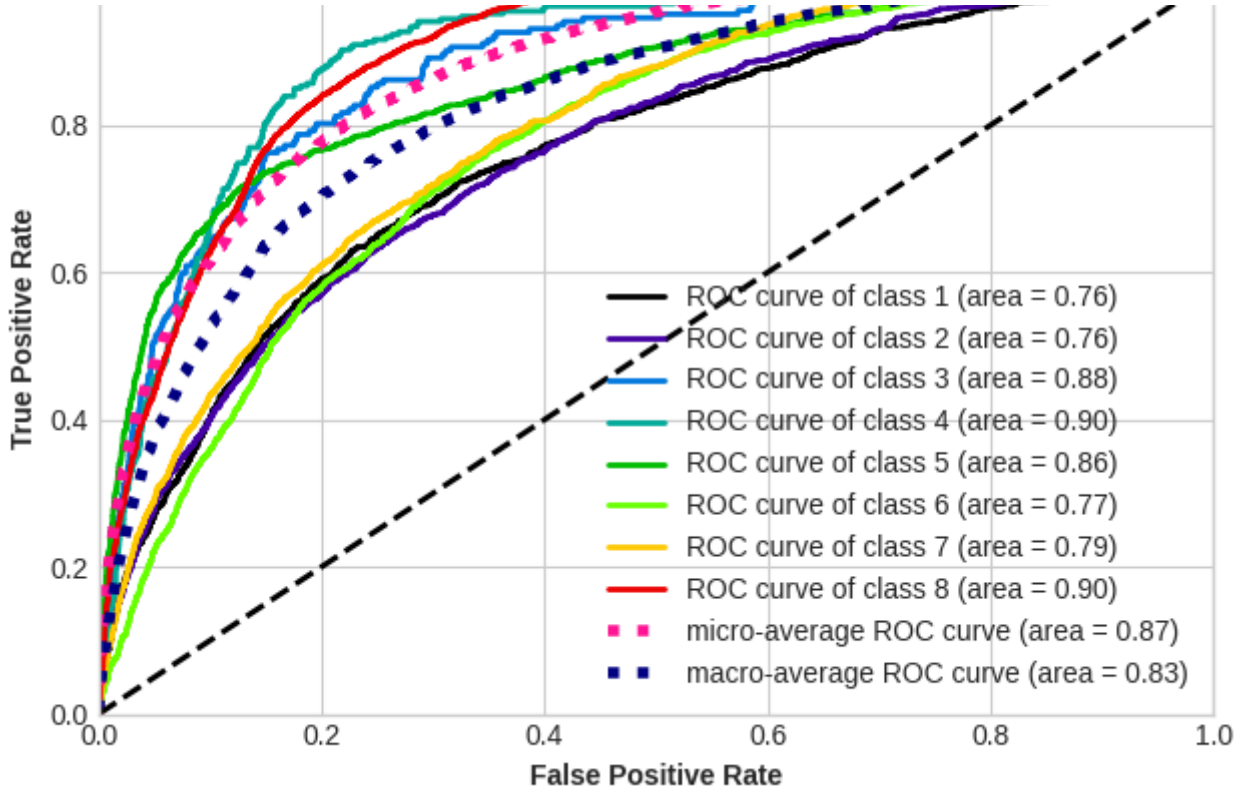

ROC Curves
True Positive Rate
False Positive Rate
ROC curve of class 1 (area = 0.70)
ROC curve of class 2 (area = 0.72)
ROC curve of class 3 (area = 0.83)
ROC curve of class 4 (area = 0.87)
ROC curve of class 5 (area = 0.84)
ROC curve of class 6 (area = 0.75)
ROC curve of class 7 (area = 0.77)
ROC curve of class 8 (area = 0.88)
micro-average ROC curve (area = 0.86)
macro-average ROC curve (area = 0.80)


ROC Curves
True Positive Rate
ROC curve of class 1 (area = 0.74)
ROC curve of class 2 (area = 0.75)
ROC curve of class 3 (area = 0.85)
ROC curve of class 4 (area = 0.88)
ROC curve of class 5 (area = 0.86)
ROC curve of class 6 (area = 0.76)
ROC curve of class 7 (area = 0.77)
ROC curve of class 8 (area = 0.89)
micro-average ROC curve (area = 0.87)
macro-average ROC curve (area = 0.81)


0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
ROC Curves
True Positive Rate
ROC curve of class 1 (area = 0.75)
ROC curve of class 2 (area = 0.75)
ROC curve of class 3 (area = 0.85)
ROC curve of class 4 (area = 0.89)
ROC curve of class 5 (area = 0.83)
ROC curve of class 6 (area = 0.72)
ROC curve of class 7 (area = 0.75)
ROC curve of class 8 (area = 0.88)
micro-average ROC curve (area = 0.85)
macro-average ROC curve (area = 0.80)


ROC Curves
True Positive Rate
False Positive Rate
ROC curve of class 1 (area = 0.78)
ROC curve of class 2 (area = 0.79)
ROC curve of class 3 (area = 0.89)
ROC curve of class 4 (area = 0.90)
ROC curve of class 5 (area = 0.86)
ROC curve of class 6 (area = 0.77)
ROC curve of class 7 (area = 0.79)
ROC curve of class 8 (area = 0.90)
micro-average ROC curve (area = 0.88)
macro-average ROC curve (area = 0.84)


ROC Curves
1.0



As shown in the charts above, each model has a wide range of AUC values against each class, indicating a varying degree of sensitivity/specificity across the dataset. The two highest average AUC values were achieved by the gradient boosting classifiers (**model 11**: macro-average AUC=0.84, and **model 12**: macro-average AUC=0.83).

It is also worth noting that both of these models performed best when predicting applicants for classes 3/4/8, as indicated by the high AUC scores (~0.9) for their respective ROC curves, whereas they perform slightly worse when classifying applicants into the remaining groups.

5.1.2 Classification Reports (Precisions/Recalls/F1-Scores)

```
# For each model, print a Classification Report to display model performance - Pre

for y_val_preds in Valid_Preds:
    print(classification_report(y_val, y_val_preds))
```

```
   macro avg       0.35      0.26      0.25     11876
weighted avg       0.43      0.46      0.38     11876

              precision    recall  f1-score   support

           1       0.47      0.18      0.26      1242
           2       0.50      0.22      0.30      1310
           3       0.00      0.00      0.00       202
           4       0.30      0.01      0.02       286
           5       0.55      0.51      0.53      1087
           6       0.39      0.56      0.46      2246
           7       0.45      0.35      0.39      1605
           8       0.64      0.88      0.74      3898

    accuracy                           0.53     11876
   macro avg       0.41      0.34      0.34     11876
weighted avg       0.51      0.53      0.49     11876

              precision    recall  f1-score   support

           1       0.42      0.23      0.29      1242
           2       0.41      0.21      0.28      1310
           3       0.45      0.05      0.09       202
           4       0.35      0.06      0.10       286
           5       0.55      0.53      0.54      1087
           6       0.41      0.53      0.46      2246
           7       0.45      0.32      0.38      1605
           8       0.64      0.87      0.74      3898

    accuracy                           0.53     11876
   macro avg       0.46      0.35      0.36     11876
weighted avg       0.50      0.53      0.50     11876

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:147
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:147
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:147
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:147
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:147
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:147
  _warn_prf(average, modifier, msg_start, len(result))
```

As shown in the summary tables above, each model has a wide range of precision/recall values against each class, indicating a varying degree of accuracy across the dataset.

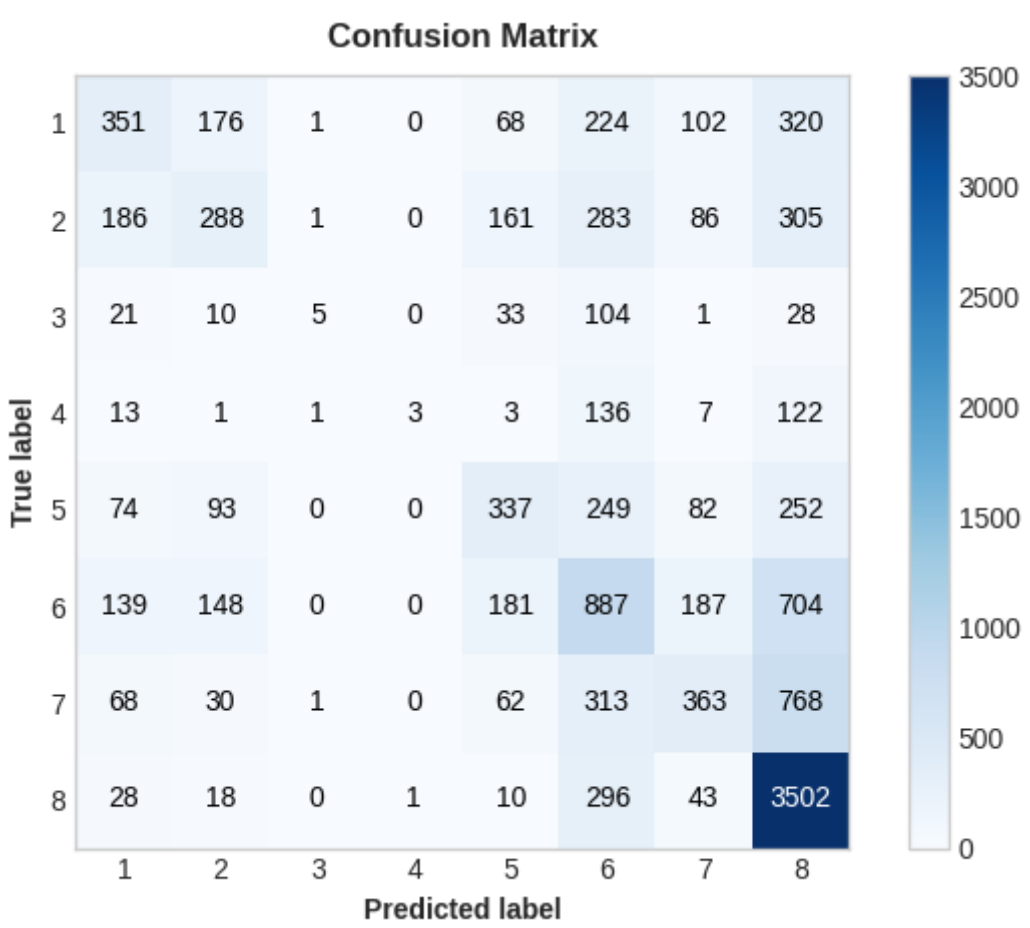
In terms of precision, each of the models display somewhat mediocre performance for each class, with the highest value consistently belonging to class 8. Each model also consistently shows very high recall for class 8, and poorer values against the other groups. The same trend can also be observed for F1-score, indicating that the models are strongly fitted towards predicting applicants in class 8 as we would expect (given that they represent the largest class in our dataset).

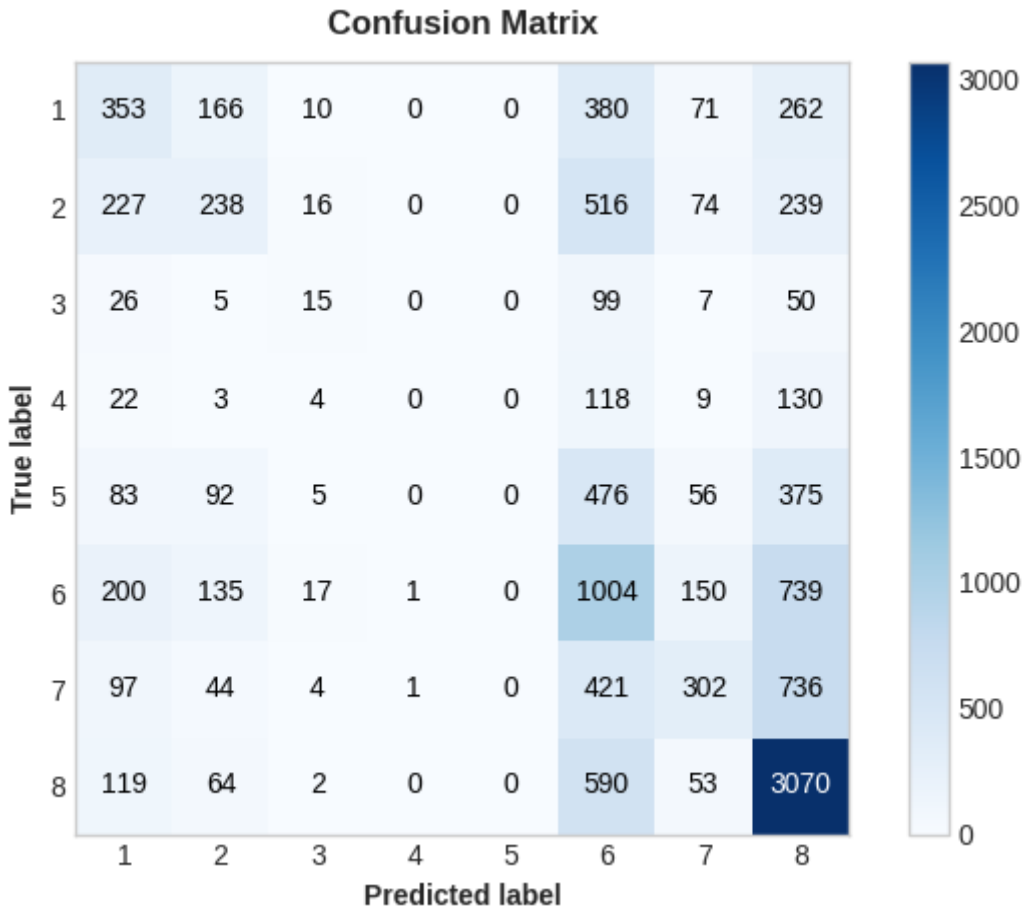
It is interesting to observe that, for class 3, **model 11**'s precision/recall/F1-score values are equal to 0, which means that it did not correctly predict any applicants for this risk rating. On the other hand, **model 12** showed a precision of 0.27, a recall of 0.05 and an F1-score of 0.08. However, as will be discussed further in the next section, this does not necessarily mean that model 11 is wholly inaccurate and should not be trusted altogether.

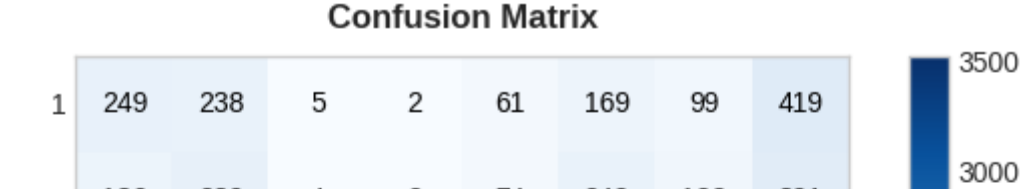
Finally, it is important not to judge the models too severely based on their precision/recall/F1-scores alone. These are typically harsh metrics that can indicate where models are strongly under/overfitting as they evaluate accuracy in a "one vs. rest" fashion - if the set of possible Response values was much smaller instead, e.g. by grouping together classes 1-8 into Low/Med/High-risk, then each model's performance would appear to significantly improve.

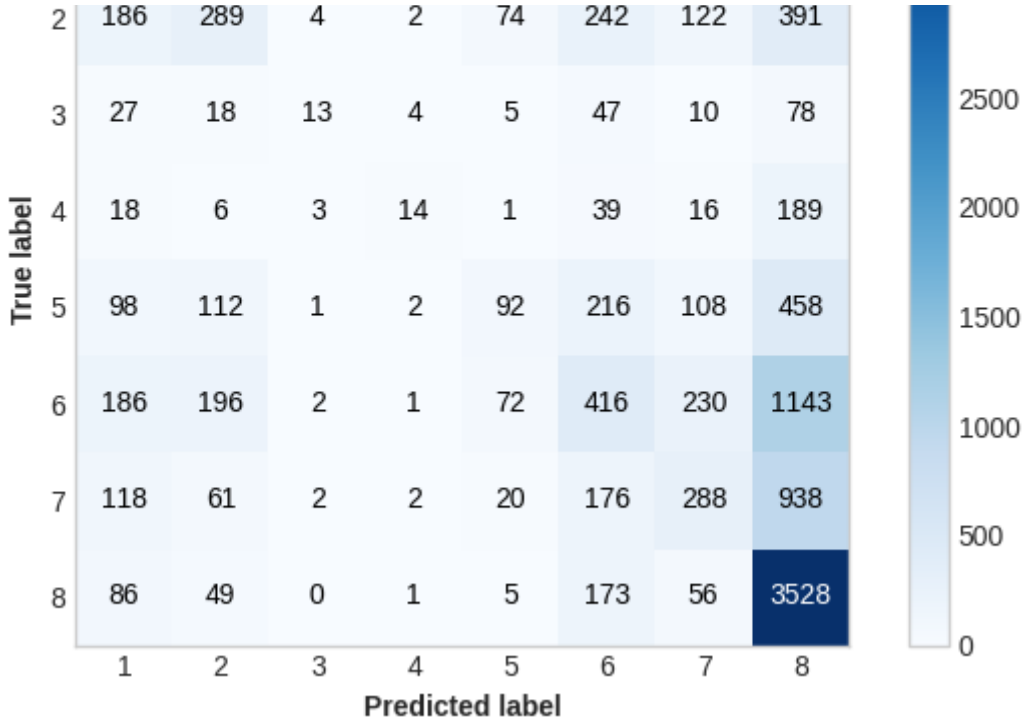
5.1.3 Confusion Matrices

```
# For each model, plot a Confusion Matrix to display model performance - True labe

for y_val_preds in Valid_Preds:
    plot_confusion_matrix(y_val, y_val_preds)
```

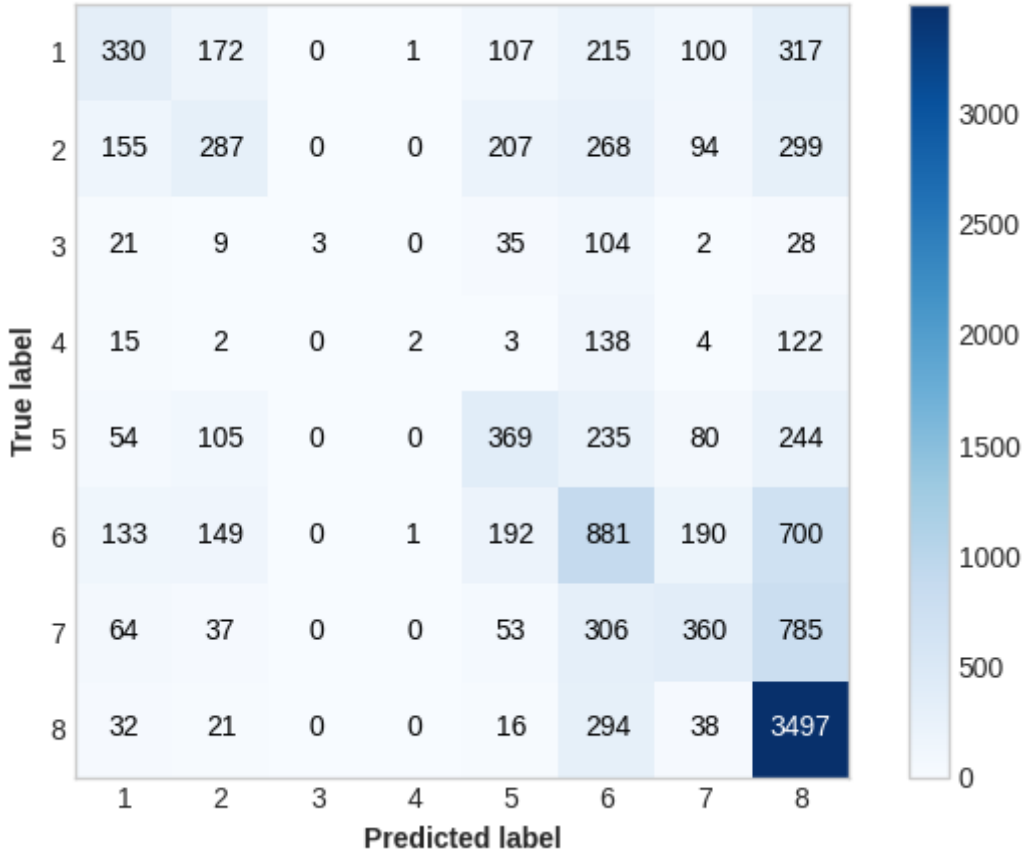

Confusion Matrix
True label
Predicted label
1 351 176 1 0 68 224 102 320
2 186 288 1 0 161 283 86 305
3 21 10 5 0 33 104 1 28
4 13 1 1 3 3 136 7 122
5 74 93 0 0 337 249 82 252
6 139 148 0 0 181 887 187 704
7 68 30 1 0 62 313 363 768
8 28 18 0 1 10 296 43 3502
1 2 3 4 5 6 7 8
3500 3000 2500 2000 1500 1000 500 0


Confusion Matrix
True label
Predicted label
1 353 166 10 0 0 380 71 262
2 227 238 16 0 0 516 74 239
3 26 5 15 0 0 99 7 50
4 22 3 4 0 0 118 9 130
5 83 92 5 0 0 476 56 375
6 200 135 17 1 0 1004 150 739
7 97 44 4 1 0 421 302 736
8 119 64 2 0 0 590 53 3070
1 2 3 4 5 6 7 8
3000 2500 2000 1500 1000 500 0

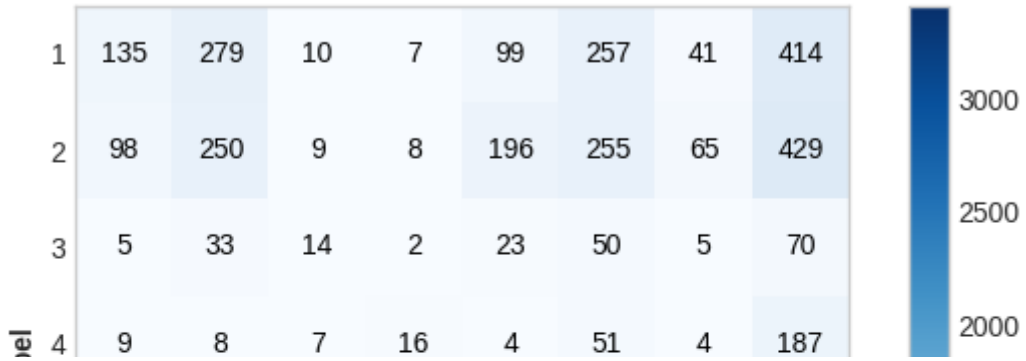

Confusion Matrix
1 249 238 5 2 61 169 99 419
3500 3000

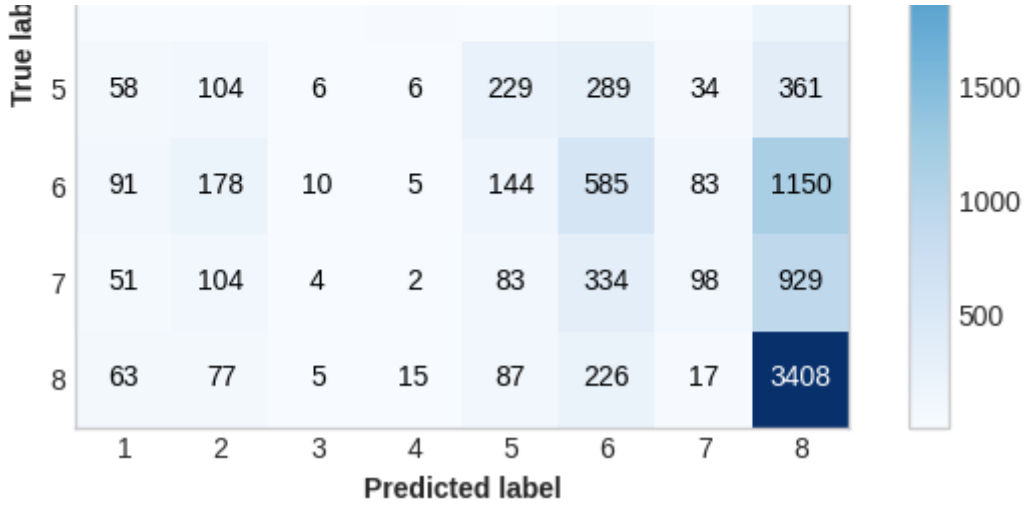


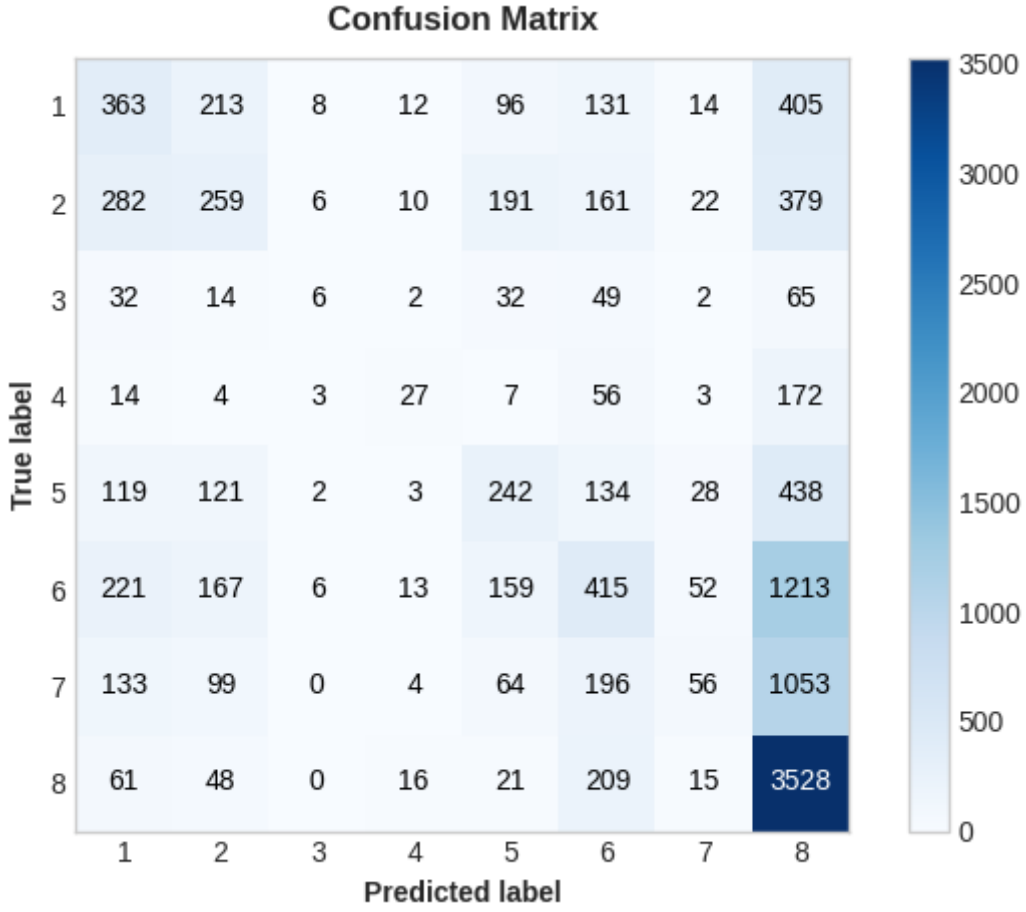
Confusion Matrix

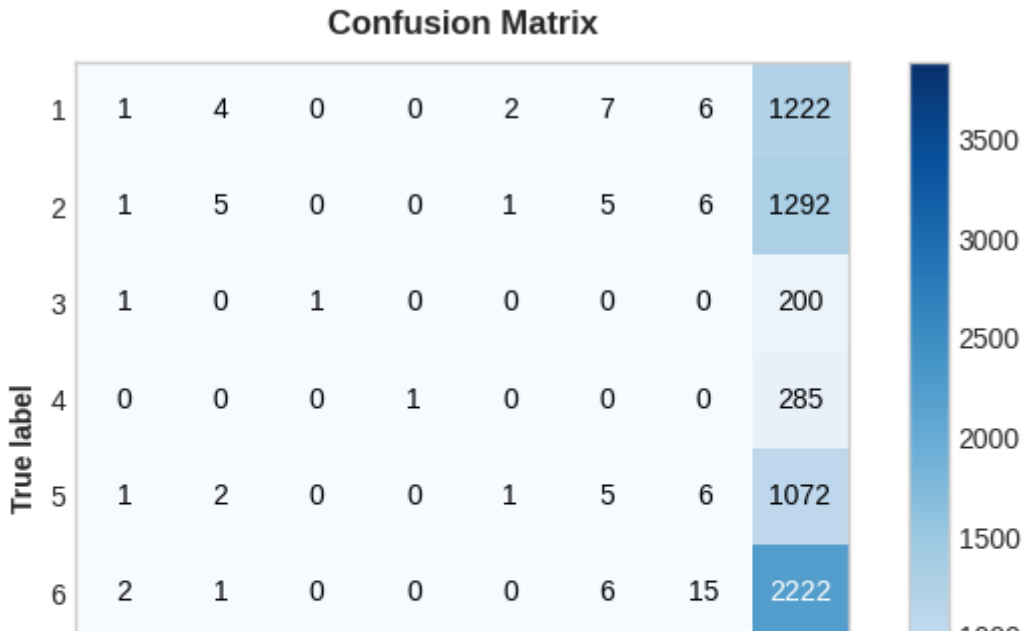


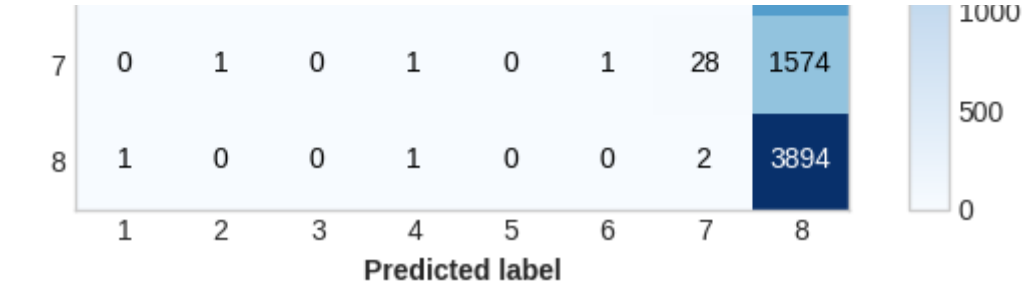
Confusion Matrix

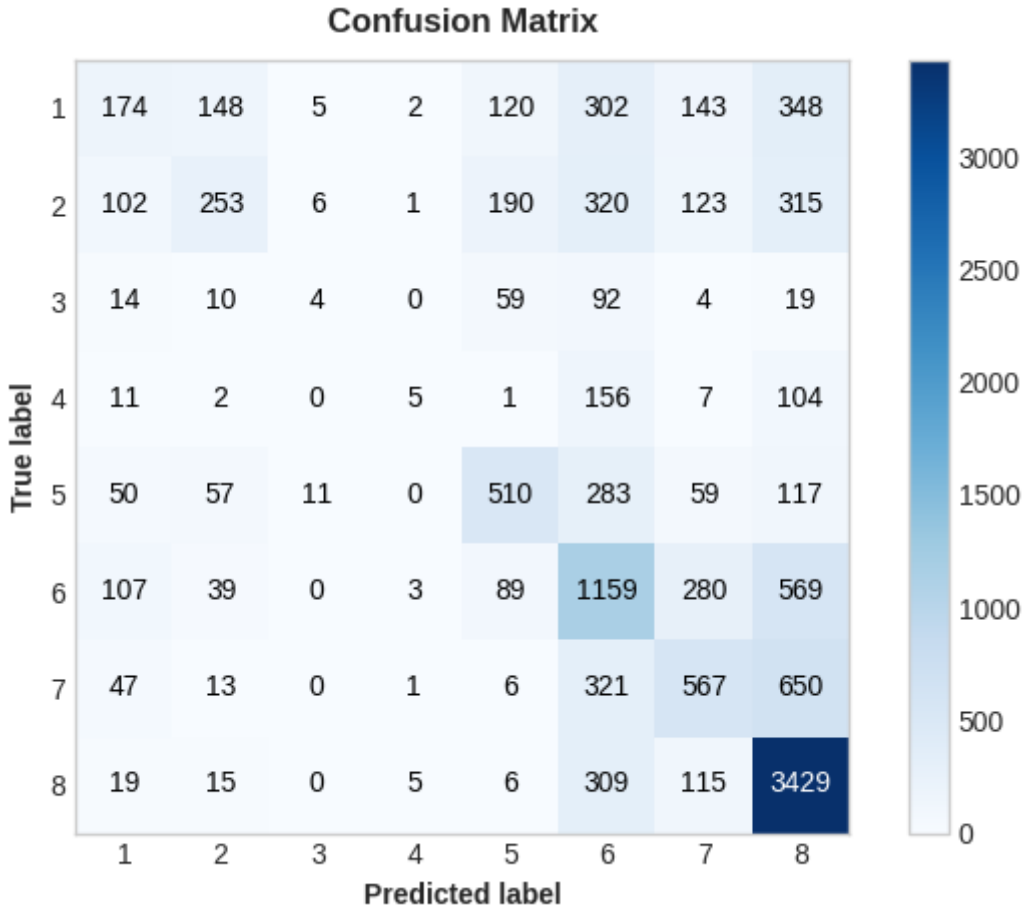


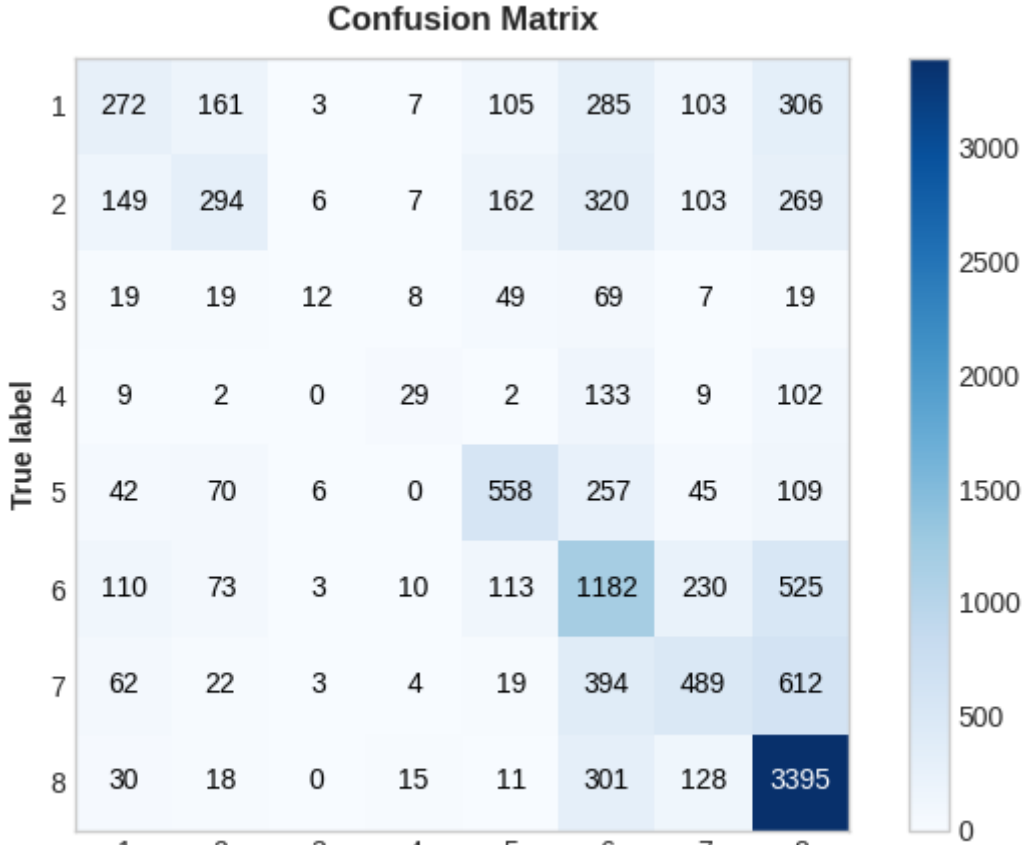

True lab
5 58 104 6 6 229 289 34 361
6 91 178 10 5 144 585 83 1150
7 51 104 4 2 83 334 98 929
8 63 77 5 15 87 226 17 3408
1 2 3 4 5 6 7 8
Predicted label
1500
1000
500

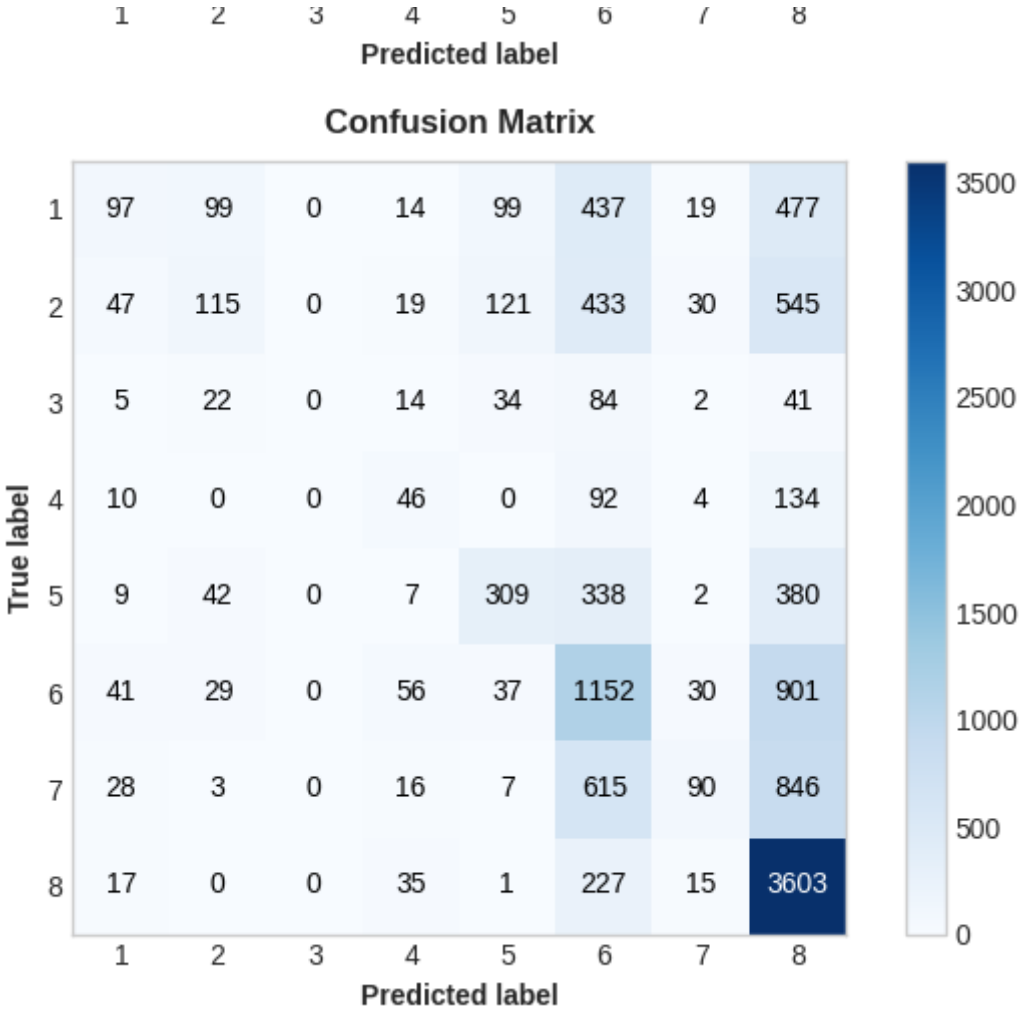

Confusion Matrix
1 363 213 8 12 96 131 14 405
2 282 259 6 10 191 161 22 379
3 32 14 6 2 32 49 2 65
4 14 4 3 27 7 56 3 172
5 119 121 2 3 242 134 28 438
6 221 167 6 13 159 415 52 1213
7 133 99 0 4 64 196 56 1053
8 61 48 0 16 21 209 15 3528
1 2 3 4 5 6 7 8
True label
Predicted label
3500
3000
2500
2000
1500
1000
500
0

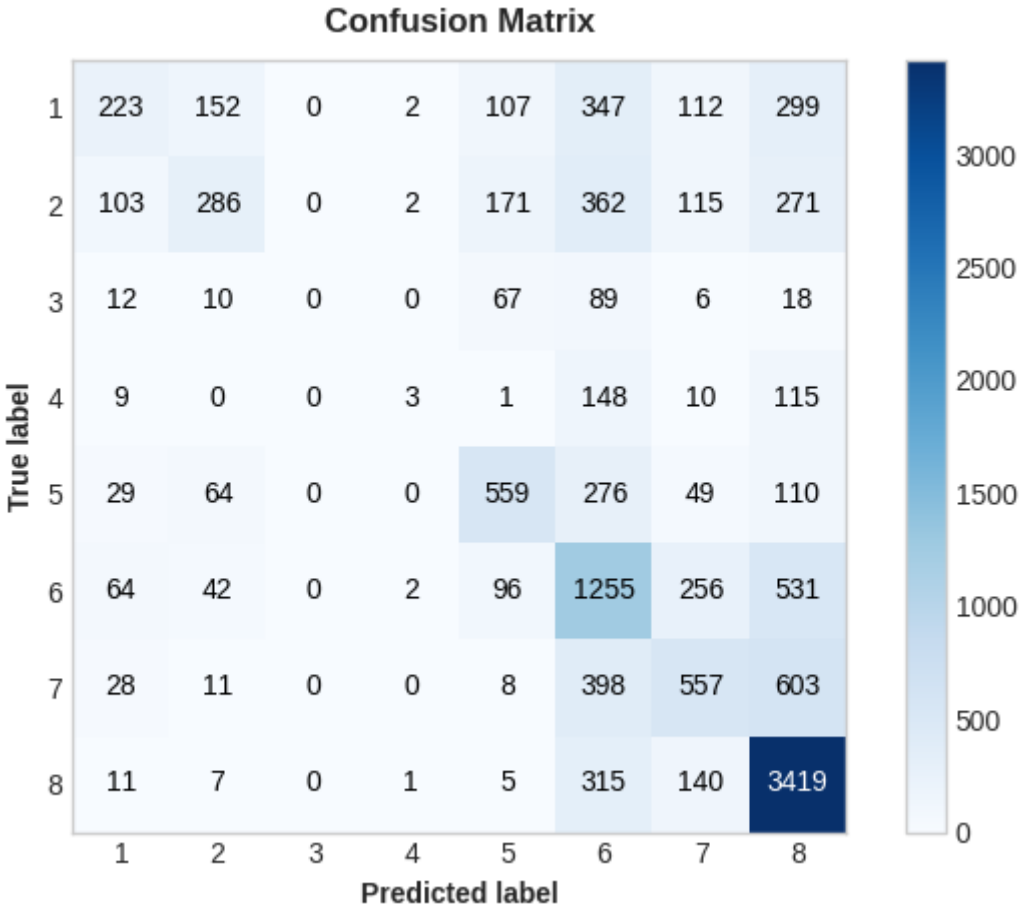

Confusion Matrix
1 1 4 0 0 2 7 6 1222
2 1 5 0 0 1 5 6 1292
3 1 0 1 0 0 0 0 200
4 0 0 0 1 0 0 0 285
5 1 2 0 0 1 5 6 1072
6 2 1 0 0 0 6 15 2222
True label
3500
3000
2500
2000
1500


7 0 1 0 1 0 1 28 1574
8 1 0 0 1 0 0 2 3894
1 2 3 4 5 6 7 8
Predicted label
1000
500
0

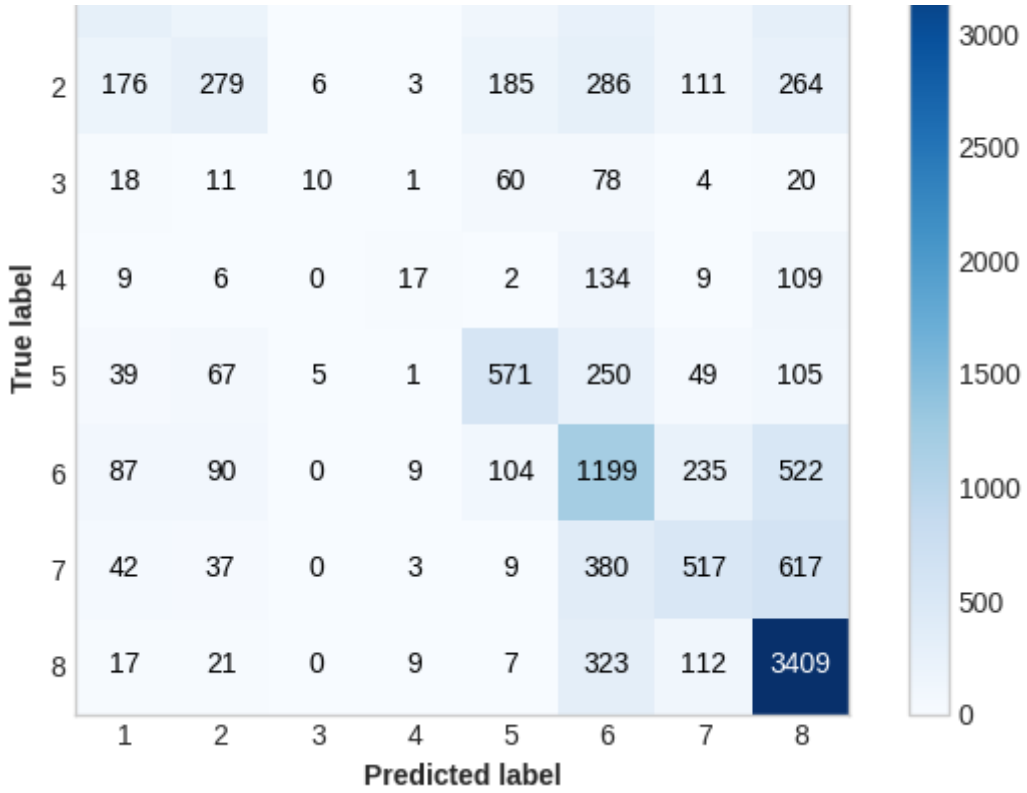

Confusion Matrix
1 174 148 5 2 120 302 143 348
2 102 253 6 1 190 320 123 315
3 14 10 4 0 59 92 4 19
4 11 2 0 5 1 156 7 104
5 50 57 11 0 510 283 59 117
6 107 39 0 3 89 1159 280 569
7 47 13 0 1 6 321 567 650
8 19 15 0 5 6 309 115 3429
1 2 3 4 5 6 7 8
Predicted label
True label
3000
2500
2000
1500
1000
500
0


Confusion Matrix
1 272 161 3 7 105 285 103 306
2 149 294 6 7 162 320 103 269
3 19 19 12 8 49 69 7 19
4 9 2 0 29 2 133 9 102
5 42 70 6 0 558 257 45 109
6 110 73 3 10 113 1182 230 525
7 62 22 3 4 19 394 489 612
8 30 18 0 15 11 301 128 3395
True label
3000
2500
2000
1500
1000
500
0


1 2 3 4 5 6 7 8
Predicted label
Confusion Matrix
1 97 99 0 14 99 437 19 477
2 47 115 0 19 121 433 30 545
3 5 22 0 14 34 84 2 41
4 10 0 0 46 0 92 4 134
5 9 42 0 7 309 338 2 380
6 41 29 0 56 37 1152 30 901
7 28 3 0 16 7 615 90 846
8 17 0 0 35 1 227 15 3603
True label
1 2 3 4 5 6 7 8
Predicted label
3500
3000
2500
2000
1500
1000
500
0


Confusion Matrix
1 223 152 0 2 107 347 112 299
2 103 286 0 2 171 362 115 271
3 12 10 0 0 67 89 6 18
4 9 0 0 3 1 148 10 115
5 29 64 0 0 559 276 49 110
6 64 42 0 2 96 1255 256 531
7 28 11 0 0 8 398 557 603
8 11 7 0 1 5 315 140 3419
True label
1 2 3 4 5 6 7 8
Predicted label
3000
2500
2000
1500
1000
500
0


Confusion Matrix
1 280 175 1 5 108 265 108 300



The confusion matrices above have been assembled in order to visualise how each model's predictions map to the true set of labels available in the test dataset. These plots help to illustrate as to whether the model has become overfitted to any particular classes in the dataset, or whether they remain sufficiently generalised and capable of sensibly classifying applicants.

From the plots above, most appear to be fairly well-fitted to the data, in that they are capable of replicating the distribution of Response values (shown earlier above in Step 1) reasonably well. We can also see that a majority of the missed cases are in fact relatively close to where they should be (e.g. predicted as class 8, was actually class 7). However, there are some notably underperforming models which appear to be highly overfitted towards predicting applicants as belonging to class 8 (which represents the largest proportion across all applicants in the dataset). The confusion matrix for **model 7** (SVC with sigmoid kernel) demonstrates this trait extremely well, which shows that only a tiny proportion of predictions were made in any other class apart from 8. **Models 3 and 5** (SVCs with linear and polynomial kernels, respectively) also demonstrate this trend to a lesser extent as well.

Models 11 and 12, on the other hand, appear to generalise reasonably well to the dataset, mimicking the distribution plot shown in Step 1. Furthermore, the only notable drawback of these models that can be observed is that they tend to predict some low-risk applicants (e.g. classes 1-2) as high-risk applicants (e.g. classes 6-8) with a higher than expected frequency - see the top-right corners of each plot. In terms of using either of these models in a real-world business scenario, this would only mean that more effort/time is potentially wasted on scrutinising low-risk applicants further before offering a policy, rather than treating high-risk applicants with a light touch and introducing unnecessary risk into the insurer's portfolio.

5.1.4 Balanced Accuracy Scores

Here, we calculate the balanced accuracy score of each model, by comparing their predictions against the known set of results in the validation dataset. This can be derived as the arithmetic mean of the recalls of each class.

It is important to note that, following similar logic to what has already been mentioned above, this metric is also harsh in that it categorises the predictions in a 'one-vs-rest' fashion - this does not adequately reflect the level of difficulty in correctly classifying each applicant with the exact rating they were given in the validation dataset.

We can derive the balanced accuracy scores of each model as follows:

```
BalancedAccScores = []

# Calculate the balanced accuracy score for each model's set of predictions.
for y_val_preds in Valid_Preds:
    BalancedAccScore = balanced_accuracy_score(y_val, y_val_preds)
    BalancedAccScores.append(BalancedAccScore)

# Collect all Balanced Accuracy scores in a single dictionary.
BalancedAccScoreResults_y_val   = {'Model1_Calibrated': BalancedAccScores[0],
                                   'Model2_Calibrated': BalancedAccScores[1],
                                   'Model3_Calibrated': BalancedAccScores[2],
                                   'Model4_Calibrated': BalancedAccScores[3],
                                   'Model5_Calibrated': BalancedAccScores[4],
                                   'Model6_Calibrated': BalancedAccScores[5],
                                   'Model7_Calibrated': BalancedAccScores[6],
                                   'Model8_Calibrated': BalancedAccScores[7],
                                   'Model9_Calibrated': BalancedAccScores[8],
                                   'Model10_Calibrated': BalancedAccScores[9],
                                   'Model11_Calibrated': BalancedAccScores[10],
                                   'Model12_Calibrated': BalancedAccScores[11]}

# Select the model with the highest Balanced Accuracy.
Model_LowestBalancedAccScore_Valid = max(BalancedAccScoreResults_y_val, key=Balanc
print(Model_LowestBalancedAccScore_Valid)
```

```
Model9_Calibrated
```

5.2 Find Best Model

Assess & determine the best performing model, based on the metrics discussed above

In the section above, we considered a variety of performance-based factors that are commonly involved in choosing the best model to use for generating a final set of predictions.

Based on the discussions above RE: ROC Curve/Confusion Matrix/Balanced Accuracy Score analyses, we select Model 12 (XGBClassifier) as the best performing model to be used for testing purposes.

```
# Designate model 12 (XGBClassifier) as the best performing model of the cohort.
BestModel = Model12_Calibrated
```

5.3 Predict using final model (using test data)

Generate predictions/probability estimates using the best model (using test data)

Using the best performing model as determined above, we can now generate a set of test predictions which will be used to evaluate/review the model's performance on unseen data.

```
# Use the best performing model to generate predictions (as probability estimates)
BestModel_Test_PredProb = BestModel.predict_proba(X_test_L1reg)

## Convert the probability estimates into label-encoded predictions, then convert
BestModel_Test_Preds = np.argmax(BestModel_Test_PredProb, axis=1)
BestModel_Test_Classes = BestModel.classes_
BestModel_Test_Preds = [BestModel_Test_Classes[i] for i in BestModel_Test_Preds]
BestModel_Test_Preds = le.inverse_transform(BestModel_Test_Preds)
```

5.4 Evaluate performance of final model (using test data)

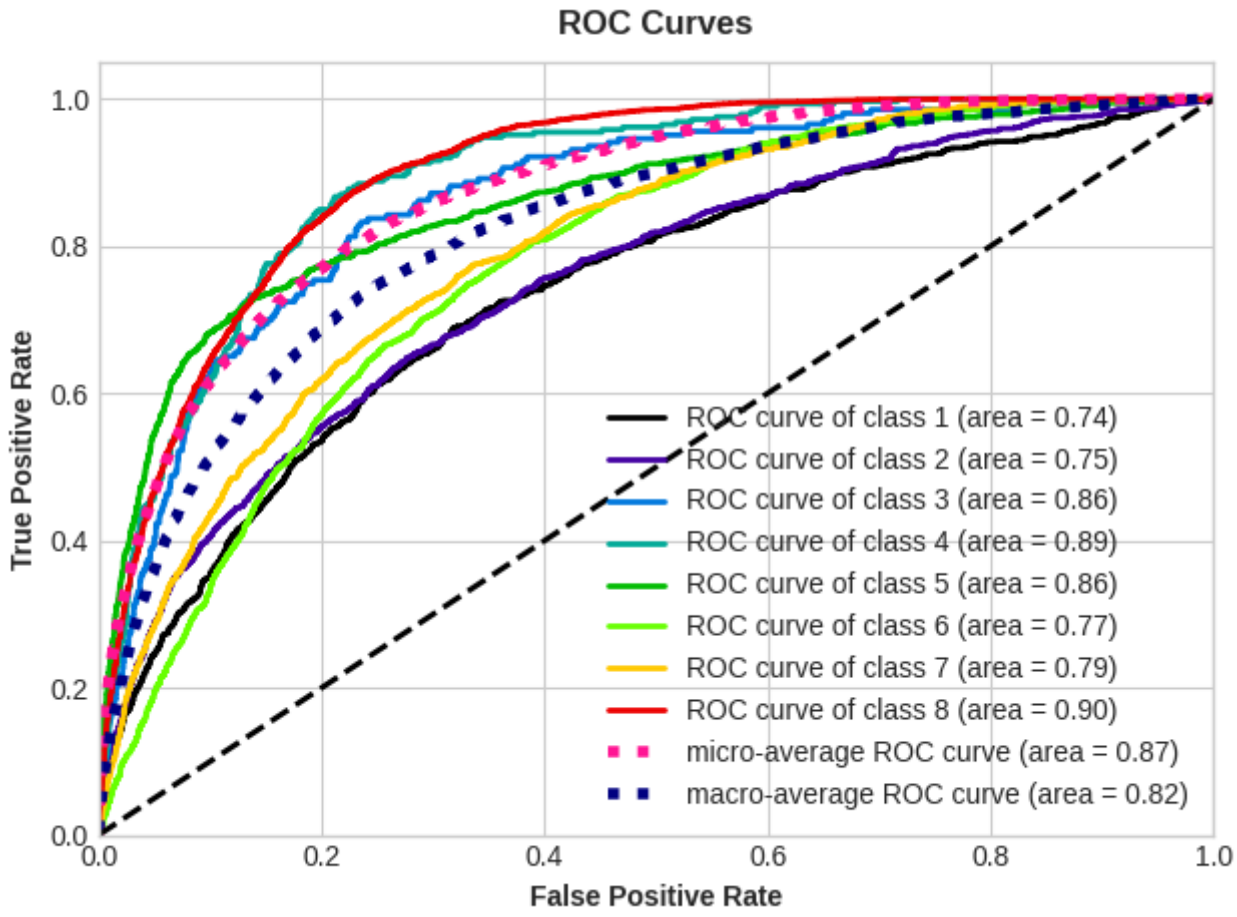
In the section below, we will review how the selected model has performed in terms of classifying applicants into each Response group, based on the test dataset.

```
# Convert the test dataset's encoded labels (0-7) back to the original set of clas
y_test = le.inverse_transform(y_test)

# Create a duplicate copy of the model's predicted labels.
Test_Preds = BestModel_Test_Preds

# Create a duplicate copy of the model's prediction probabilities.
Test_PredProbs = BestModel_Test_PredProb
```

5.4.1 ROC Curve

```
# Plot the ROC curve to display the selected model's performance - TP rate vs FP r
plot_roc(y_test, Test_PredProbs)
```

```
<Axes: title={'center': 'ROC Curves'}, xlabel='False Positive Rate',
ylabel='True Positive Rate'>
```



As shown in the chart above, the `XGBClassifier` model features a similarly broad range of AUC values against each class, indicating a varying degree of sensitivity/specificity across the dataset. Furthermore, the model has also performed relatively strongly when predicting applicants for classes 3/4/8, as indicated by the high AUC scores (~0.9) for their respective ROC curves - this is a good sign that our model is still able to generalise well, even when handling previously unseen data.

5.4.2 Classification Report (i.e. Precision/Recall/F1-Score)

```
# Print a Classification Report to display the selected model's performance - Prec

print(classification_report(y_test, Test_Preds))
```

```
              precision    recall  f1-score   support

           1       0.40      0.19      0.26      1241
           2       0.42      0.22      0.29      1310
           3       0.27      0.03      0.06       203
           4       0.35      0.07      0.11       286
           5       0.53      0.52      0.53      1086
           6       0.41      0.53      0.46      2247
           7       0.46      0.33      0.39      1606
           8       0.64      0.88      0.74      3898
```

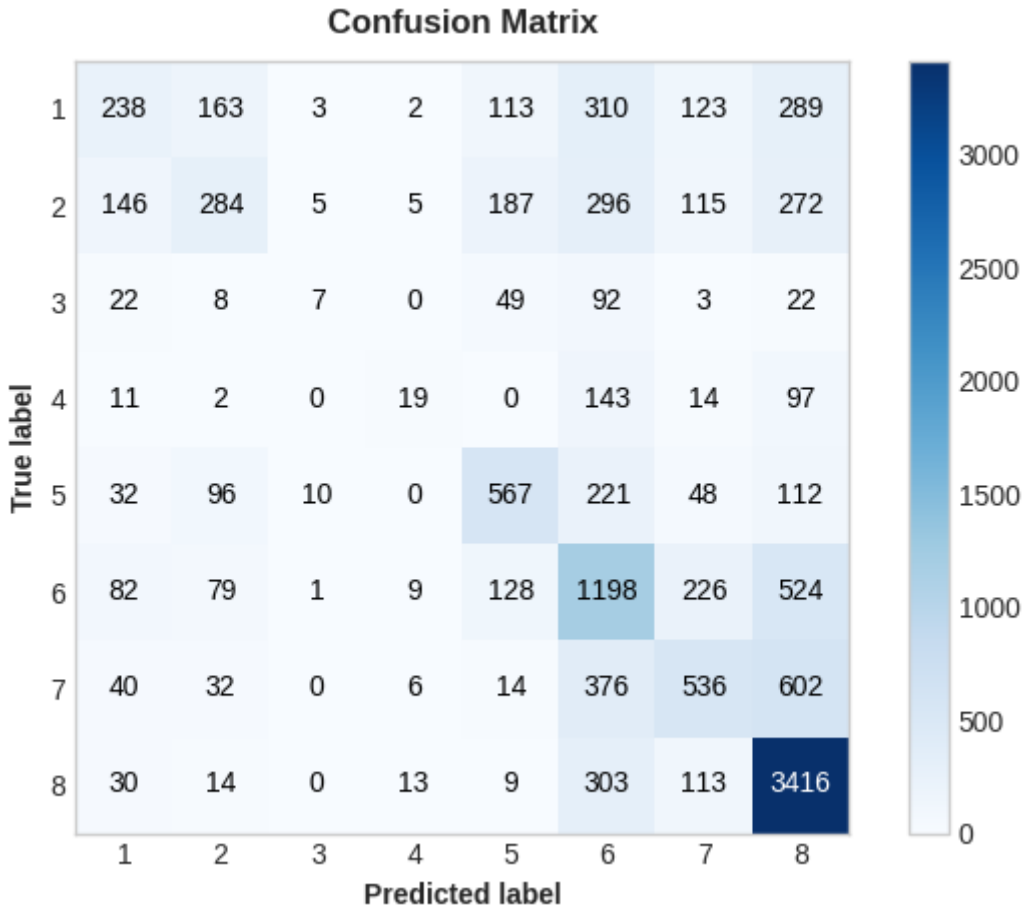
```
              accuracy                           0.53     11877
             macro avg       0.43      0.35      0.35     11877
          weighted avg       0.50      0.53      0.49     11877
```

As shown in the classification report above, the model shows a broad range of precision/recall/F1-score values against each class, indicating a varying degree of accuracy across the dataset and emphasising how the model has been generally well-fitted towards the most important Response groups. The model still appears to perform best when predicting for class 8 applicants, as evidenced by the relatively higher precision/recall/F1-score values for this group.

5.4.3 Confusion Matrix

```
# Plot a Confusion Matrix to display the selected model's performance - True label

plot_confusion_matrix(y_test, Test_Preds)
```

```
<Axes: title={'center': 'Confusion Matrix'}, xlabel='Predicted label',
ylabel='True label'>
```



From the confusion matrix above, it is clear that the model has performed on the test dataset very similarly to the validation scenario discussed earlier above. Furthermore, the model has also been relatively successful in replicating the Response distribution shown earlier in Step 1,

Any off-diagonal hotspots (representing notable proportions of incorrect predictions) have already been revealed as part of the model validation process earlier above - so this should not come as a huge surprise. The important thing to note is that, with the exception of the observations in the top-right corner (where the model has predicted higher risk levels than required), the model appears to be well-fitted towards handling a multitude of applicants with varying levels of risk.

5.4.4 Balanced Accuracy Score

Finally, we can calculate the balanced accuracy score of the selected model, using the test dataset:

```
# Calculate the balanced accuracy score for the selected model.
BalancedAccScore_y_test = balanced_accuracy_score(y_test, Test_Preds)

print(BalancedAccScore_y_test)
```

```
0.3468551367113548
```

6. XAI

Review model performance in the context of "feature importance" and explainability.

In the section above, we considered how well the model has performed against a number of key metrics such as ROC-AUC scores and precisions/recalls. However, we also need to think about how the model's decisions/behaviour can be explained in a clear and consistent manner - this is important in a real-world scenario from both a legal and ethical perspective, as we are assessing the prospective risk levels of numerous applicants for life insurance policies. Agreeing/rejecting applicants on insufficient/incorrect grounds could otherwise lead to reputational/financial damages.

In this section, we will now consider:

- what features have the biggest impact on the `XGBClassifier` model's predictions, and
- how the `XGBClassifier` model has generated its predictions based on the values of the most important features.

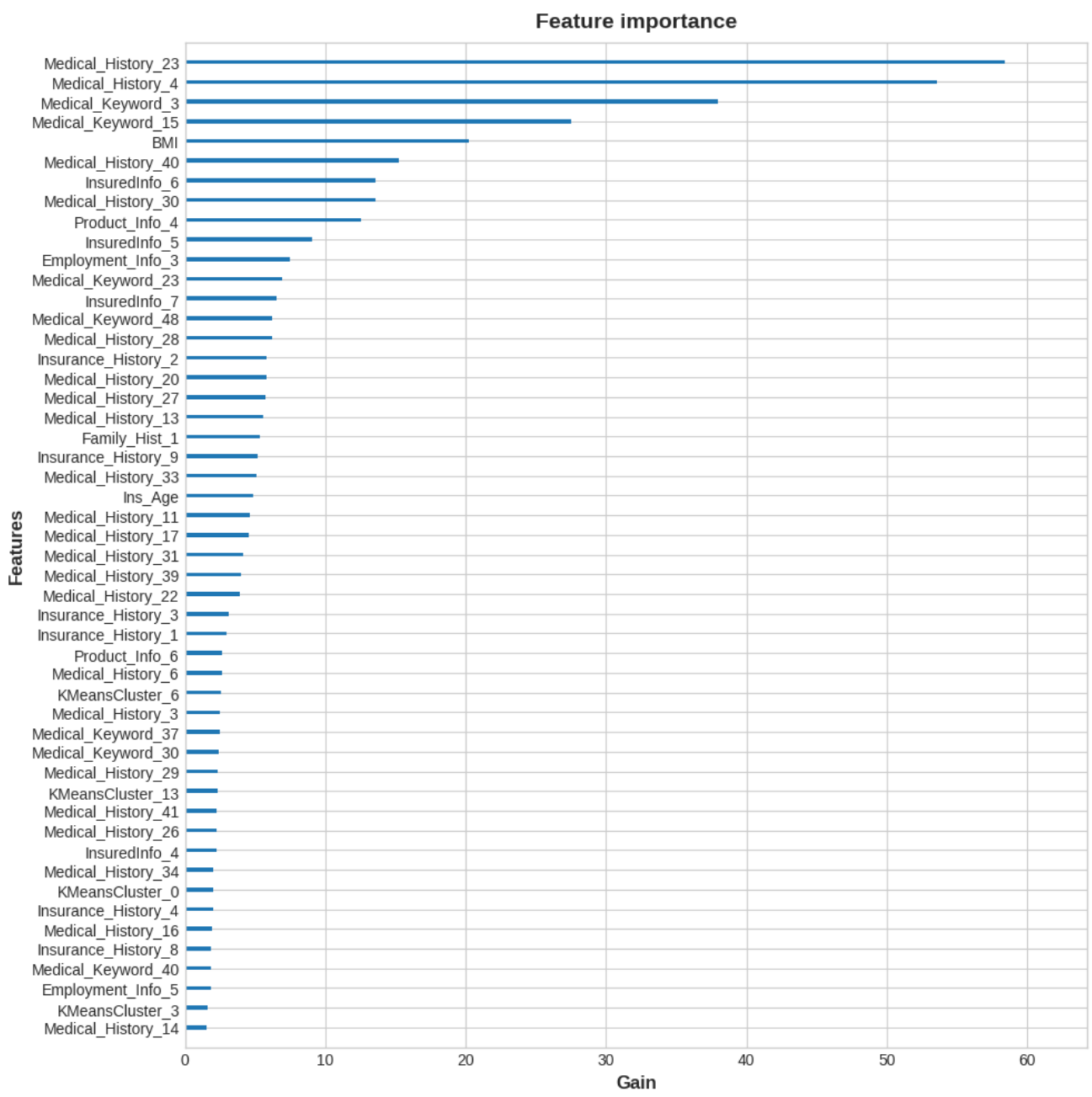
6.1 Feature Importances

Firstly, we will take a look at the XGBClassifier a bit more closely, in order to understand what features the model values most highly.

```
# Review the feature importances of the XGBoost classifier.

FeatureImportances_BestModel = BestModel.estimator.feature_importances_
print(FeatureImportances_BestModel)
```

```
[0.03071852 0.00643526 0.01189804 0.04943389 0.01821095 0.00449573
 0.00538691 0.02208213 0.03325504 0.01596296 0.00728304 0.01413486
 0.00766465 0.004864   0.00457755 0.01263332 0.01299413 0.00609502
 0.13119808 0.00642483 0.01129519 0.01367998 0.0037234  0.00468396
 0.01110481 0.0141161  0.00953209 0.14302751 0.00543216 0.01394896
 0.01512968 0.00560046 0.03314825 0.01020407 0.01248888 0.00492357
 0.00977467 0.03728448 0.00549354 0.09296282 0.06742027 0.01694823
 0.00587843 0.00598483 0.00456115 0.01524989 0.00491214 0.0038409
 0.00630035 0.00560034]
```

```
from xgboost import plot_importance

# Generate a Feature Importance plot using the selected model.

fig, ax = plt.subplots(figsize=(10, 10))
plot_importance(BestModel.estimator,
                importance_type="gain",
                xlabel="Gain",
                show_values=False,
                ax=ax)
plt.show()
```


Feature importance
Medical_History_23
Medical_History_4
Medical_Keyword_3
Medical_Keyword_15
BMI
Medical_History_40
InsuredInfo_6
Medical_History_30
Product_Info_4
InsuredInfo_5
Employment_Info_3
Medical_Keyword_23
InsuredInfo_7
Medical_Keyword_48
Medical_History_28
Insurance_History_2
Medical_History_20
Medical_History_27
Medical_History_13
Family_Hist_1
Insurance_History_9
Medical_History_33
Ins_Age
Medical_History_11
Medical_History_17
Medical_History_31
Medical_History_39
Medical_History_22
Insurance_History_3
Insurance_History_1
Product_Info_6
Medical_History_6
KMeansCluster_6
Medical_History_3
Medical_Keyword_37
Medical_Keyword_30
Medical_History_29
KMeansCluster_13
Medical_History_41
Medical_History_26
InsuredInfo_4
Medical_History_34
KMeansCluster_0
Insurance_History_4
Medical_History_16
Insurance_History_8
Medical_Keyword_40
Employment_Info_5
KMeansCluster_3
Medical_History_14
Features
0
10
20
30
40
50
60
Gain

The Gain chart above demonstrates the relative contribution of each feature to the model - which is calculated by taking each feature's contribution for each decision tree that is used in the model.

This plot shows that the top five most important features (in terms of Gain/model contribution) are:

- `Medical_History_23`,
- `Medical_History_4`,
- `Medical_Keyword_3`,
- `Medical_Keyword_15`, and
- `BMI`.

Interestingly, these five features were also highly ranked within the Mutual Information score chart (see Step 7 earlier above), which lends some credibility to the view that these features may be closely involved in governing what risk rating an applicant should be assigned.

The bottom five (i.e. least important) features in terms of Gain are:

- `Medical_History_34`,
- `Product_Info_2_E1`,
- `KMeansCluster_4`,
- `Insurance_History_8`, and
- `Medical_History_41`.

Four of these five features were also poorly ranked within the MI score chart shown in Step 7, however KMeansCluster_4 was instead moderately ranked (residing within the top 15 features). This implies that, whilst `KMeansCluster_4` was initially deemed to show some potential in terms of predictive power, the `XGBClassifier` does not value this feature as highly when generating predictions for the test dataset.

6.2 Permutation Importance

Next, we can review how the model handles random column-wise reordering of data - performing what is more commonly known as permutation importance analysis - to see which features increase the volatility of our model's predictions upon random shuffling (and hence, which features the model relies on most heavily for generating predictions).

To do this, we will calculate the permutation importance weights using our test dataset, as shown below:

```
# Downgrade scikit-learn and install eli5
!pip install --upgrade scikit-learn
import sklearn
print(sklearn.__version__)
```

```
#!pip install --upgrade seaborn
#!pip install scikit-learn==1.2.2
#!pip install scikit-learn==0.22.2.post1
!pip install --upgrade eli5
import eli5
print(eli5.__version__)
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.10/dist
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.
1.3.2
Requirement already satisfied: eli5 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: attrs>17.1.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: jinja2>=3.0.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: numpy>=1.9.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: scikit-learn>=0.20 in /usr/local/lib/python3.10
Requirement already satisfied: graphviz in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: tabulate>=0.7.7 in /usr/local/lib/python3.10/di
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-127-5bde33371f49> in <cell line: 9>()
      7 #!pip install scikit-learn==0.22.2.post1
      8 get_ipython().system('pip install --upgrade eli5')
----> 9 import eli5
     10 print(eli5.__version__)

                                3 frames
/usr/local/lib/python3.10/dist-
packages/eli5/sklearn/permutation_importance.py in <module>
      5 import numpy as np
      6 from sklearn.model_selection import check_cv
----> 7 from sklearn.utils.metaestimators import if_delegate_has_method
      8 from sklearn.utils import check_array, check_random_state
      9 from sklearn.base import (

ImportError: cannot import name 'if_delegate_has_method' from
'sklearn.utils.metaestimators' (/usr/local/lib/python3.10/dist-
packages/sklearn/utils/metaestimators.py)

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES

Next steps: Explain error

```
from eli5.sklearn import PermutationImportance
from eli5 import show_weights

# Calculate the Permutation Importances of the selected model.

perm = PermutationImportance(BestModel.estimator, random_state=0).fit(X_test_L1reg
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-129-23f7b759d0cf> in <cell line: 1>()
----> 1 from sklearn import PermutationImportance
      2 from eli5 import show_weights
      3
      4 # Calculate the Permutation Importances of the selected model.
      5

ImportError: cannot import name 'PermutationImportance' from 'sklearn'
(/usr/local/lib/python3.10/dist-packages/sklearn/__init__.py)

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES

Next steps: Explain error

```
# Show the Permutation Importance weights (plus errors) of the top 20 features.

show_weights(perm, feature_names=X_test_L1reg.columns.tolist())
```

The values towards the top of the table are the most important, and those towards the bottom matter the least.

Here, the top five features that appear to have the strongest impact on the model's performance/accuracy when shuffled randomly are:

- `Medical_Keyword_3`
- `Medical_History_39`
- `InsuredInfo_5`
- `Product_Info_2_D1`
- `Medical_History_17`

Most of these features were also highlighted as having the highest feature importances in the section above. Interestingly, `Medical_History_23` (the highest scored feature in terms of

Feature Importance) does not show up in the top 20 permutation importance weightings - this could imply that this feature does not have any meaningful causational relationship with Response, and has unintentionally been given higher importance due to possible overfitting.

More information on the topic of comparing tree-based feature importances against permutation importances can be found at the following page: Relation to impurity-based importance in trees - scikit-learn.

6.2.1 SHAP Values

The SHAP framework (acronym derived from **SH**apley **A**dditive ex**P**lanations) is excellent for breaking down predictions in order to show the impact of each feature. This is especially useful for when we want to explain how/why we have classified certain applicants who may have demonstrated high/low risk potential.

The advantage that this approach has over calculating permutation importances alone, is that SHAP values can express whether a feature has a broad effect across all predictions, or whether its effect is more localised for a handful of predictions and negligible in general; permutation importance simply captures the "average" impact of each feature.

Here, we will calculate the SHAP values for the test dataset to visualise how the XGBClassifier model has behaved, based on the values of the most important features.

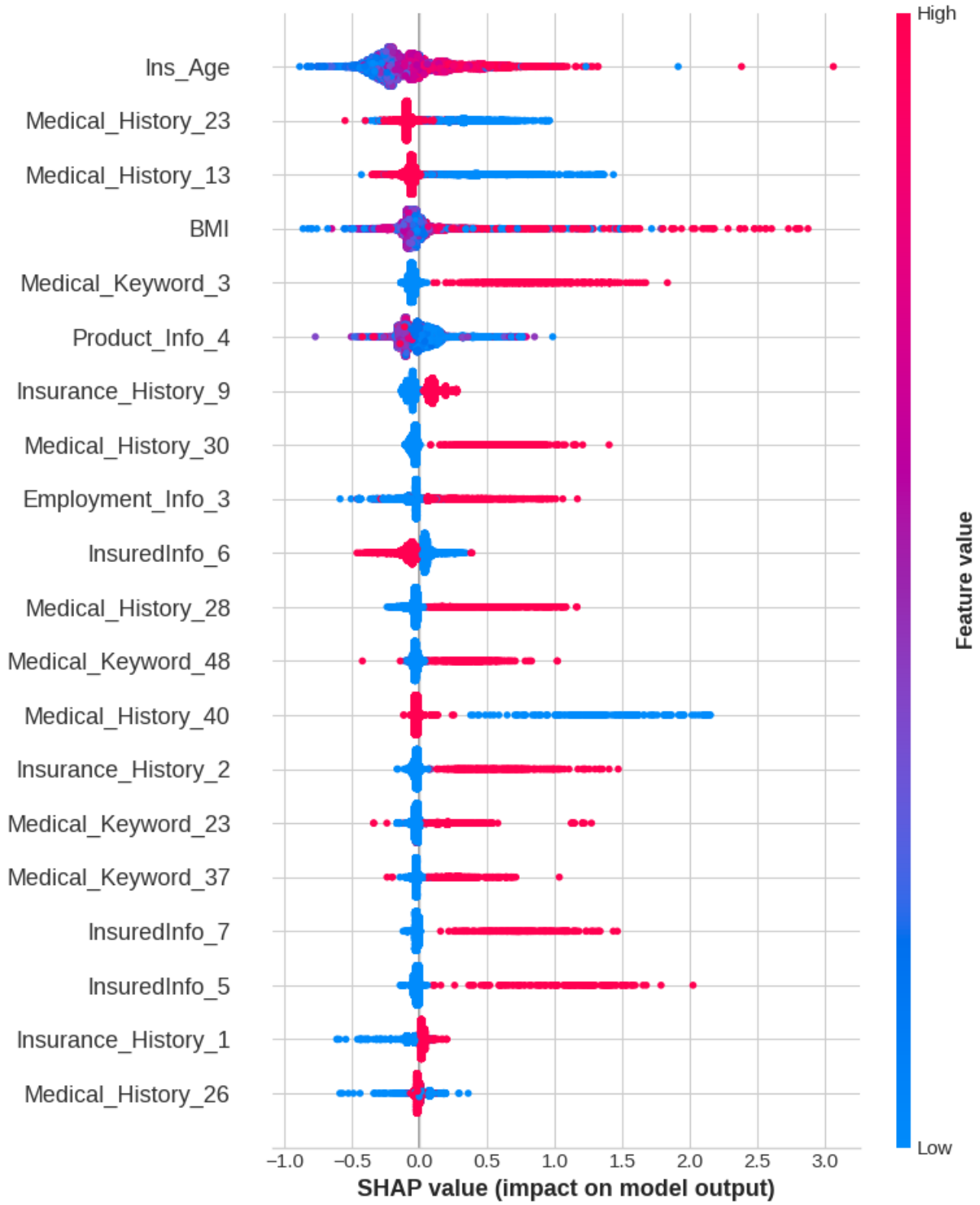
```
!pip install shap
```

```
Collecting shap
  Downloading shap-0.46.0-cp310-cp310-manylinux_2_12_x86_64.manylinux2010_x86_
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: tqdm>=4.27.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: packaging>20.9 in /usr/local/lib/python3.10/dis
Collecting slicer==0.0.8 (from shap)
  Downloading slicer-0.0.8-py3-none-any.whl.metadata (4.0 kB)
Requirement already satisfied: numba in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: cloudpickle in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/py
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-pack
Downloading shap-0.46.0-cp310-cp310-manylinux_2_12_x86_64.manylinux2010_x86_64
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 540.1/5
Downloading slicer-0.0.8-py3-none-any.whl (15 kB)
Installing collected packages: slicer, shap
Successfully installed shap-0.46.0 slicer-0.0.8
```

```
import shap

# from above...
### Designate model 12 (XGBClassifier) as the best performing model of the cohort.
### BestModel = Model12_Calibrated

# Initialise the explainer object in order to calculate SHAP values.
explainer = shap.TreeExplainer(BestModel.estimator)

# Calculate SHAP values for the whole test dataset.
shap_values = explainer.shap_values(X_test_L1reg)
```

Double-click (or enter) to edit

```
# Create a summary plot of the SHAP values.
# shap.summary_plot(shap_values, X_test_L1reg)

# Select the SHAP values for the first class/cluster if necessary
shap_values_class0 = shap_values[:,:,0]

# Create a summary plot for the first class/cluster
shap.summary_plot(shap_values_class0, X_test_L1reg)
```



The SHAP summary plot above provides us with a top-down view of feature importance, for the top 20 features as calculated via the SHAP framework. The colour of each dot represents

whether that feature was high or low (for that row in the dataset), and its horizontal location shows whether the effect of that value caused a higher (towards 8) or lower (towards 1) prediction.

For instance, the `BMI` feature clearly expresses that as the BMI of the applicant increases, then their predicted risk rating also increases strongly. As another example, `Product_Info_2_A6` - i.e. whether the applicant's selection for `Product_Info_2` is equal to *A6* - shows a clear negative correlation with the predicted risk rating.

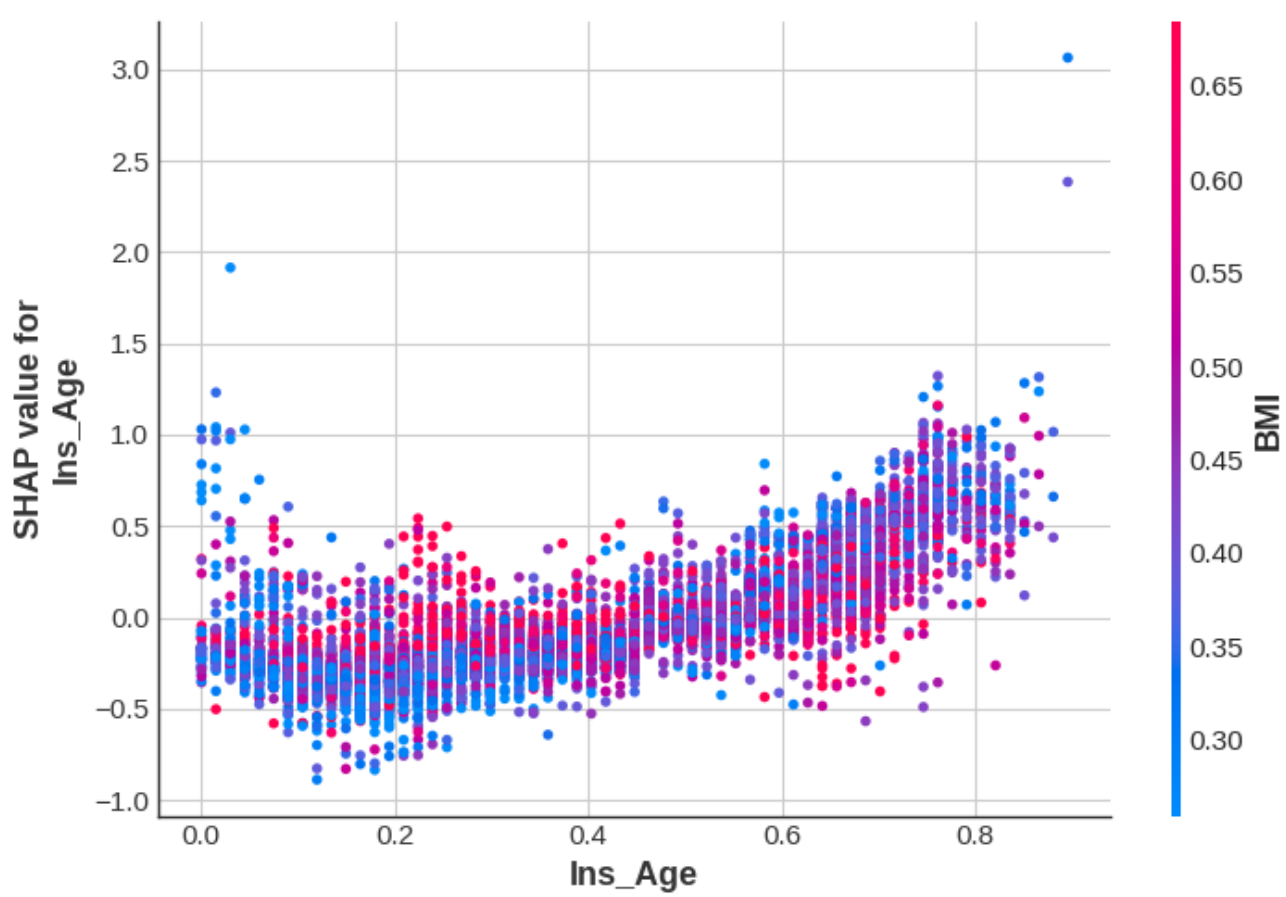
However, there are also a couple of other interesting cases to note - for example, `Ins_Age` shows a negative correlation when away from the baseline value (class 5) but is more mixed as the value approaches the baseline. `Family_Hist_4` appears to show a mixed effect on the value of `Response` regardless of the choice of class.

6.2.2 SHAP Dependence Plots

We can also review individual features more closely, by creating SHAP dependence contribution plots.

These are extremely helpful for displaying what the distribution of effects is, and linking this to a model's predictions.

```
# Create a SHAP Dependence plot of the 2nd highest ranked feature - "Product_Info_

shap.dependence_plot("rank(0)", shap_values[:, :, 0], X_test_L1reg)
```

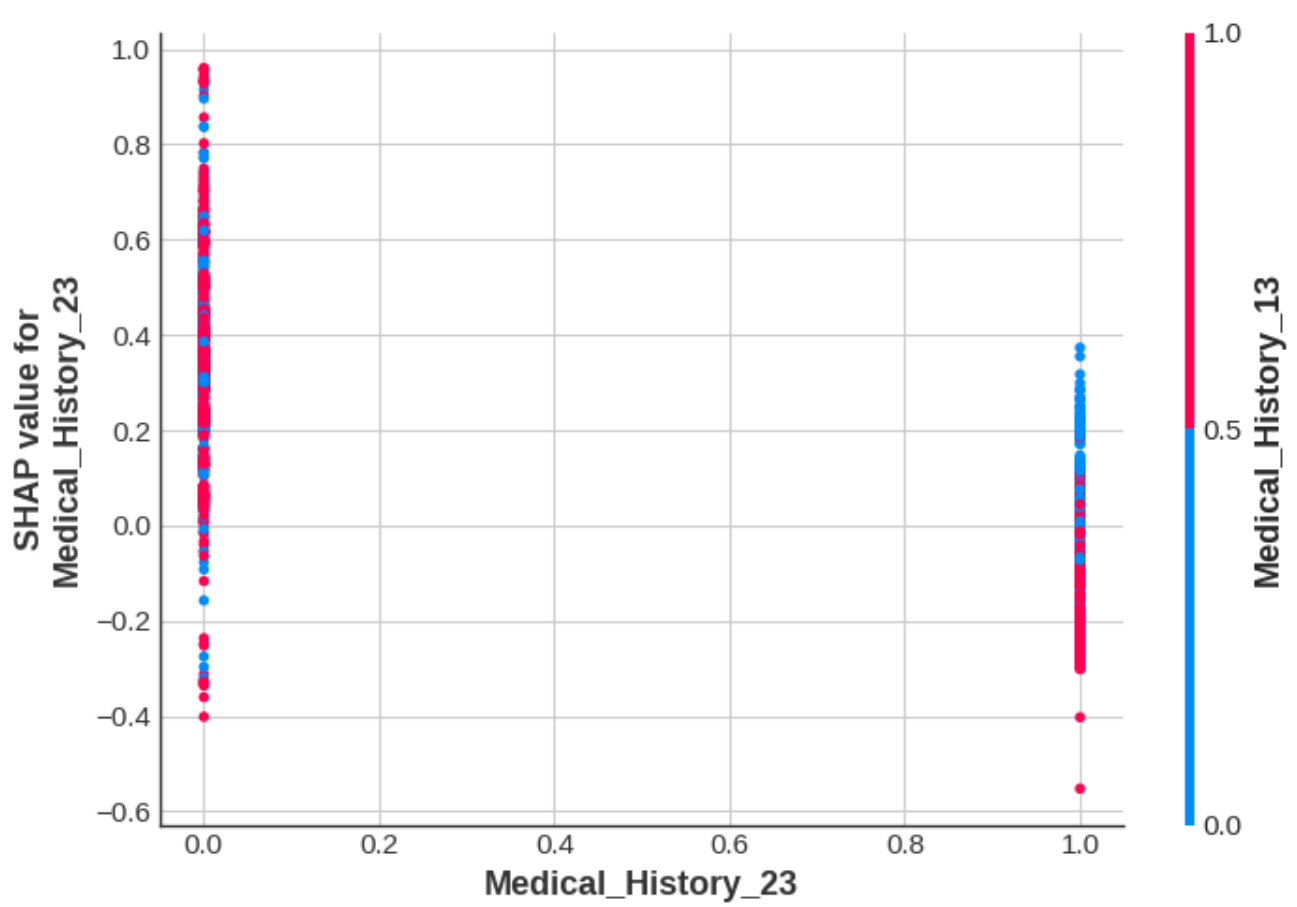


Here, we have generated a SHAP dependence contribution plot for `BMI` as a function of `Ins_Age`. The horizontal location of each dot represents the actual value from the test dataset, and the vertical location represents the impact (on the prediction) of having that value.

The shape of the plot indicates that, once BMI exceeds 0.7, the distribution slopes sharply upwards - inferring that the likelihood of being assigned to a higher risk rating greatly increases after this point. However, as `BMI` approaches 1.0, the distribution then quickly slopes downwards. The drastic increase in the distribution's spread as the feature value approaches 1.0 implies that other features begin to interact more strongly with `BMI` at this upper limit.

Upon reviewing the interaction between `BMI` and `Ins_Age` more closely at this upper limit, it appears that on average, younger applicants with high BMIs tend to be assigned higher risk ratings by the model than high-BMI applicants that are more elderly, however more data would be required in order to validate this hypothesis, before any relevant business decisions could be made.

```
shap.dependence_plot("rank(1)", shap_values[:, :, 0], X_test_L1reg)
```



Here, we generate a SHAP dependence contribution plot to analyze the influence of Medical_History_23 on the model's risk classification predictions, with Medical_History_13 as the interaction feature.

The horizontal axis represents the values of Medical_History_23, while the vertical axis displays the SHAP values for Medical_History_13. Positive SHAP values indicate that the corresponding Medical_History_23 values increase the likelihood of classification into a certain risk category, while negative SHAP values suggest a decrease. SHAP values near zero indicate minimal impact.

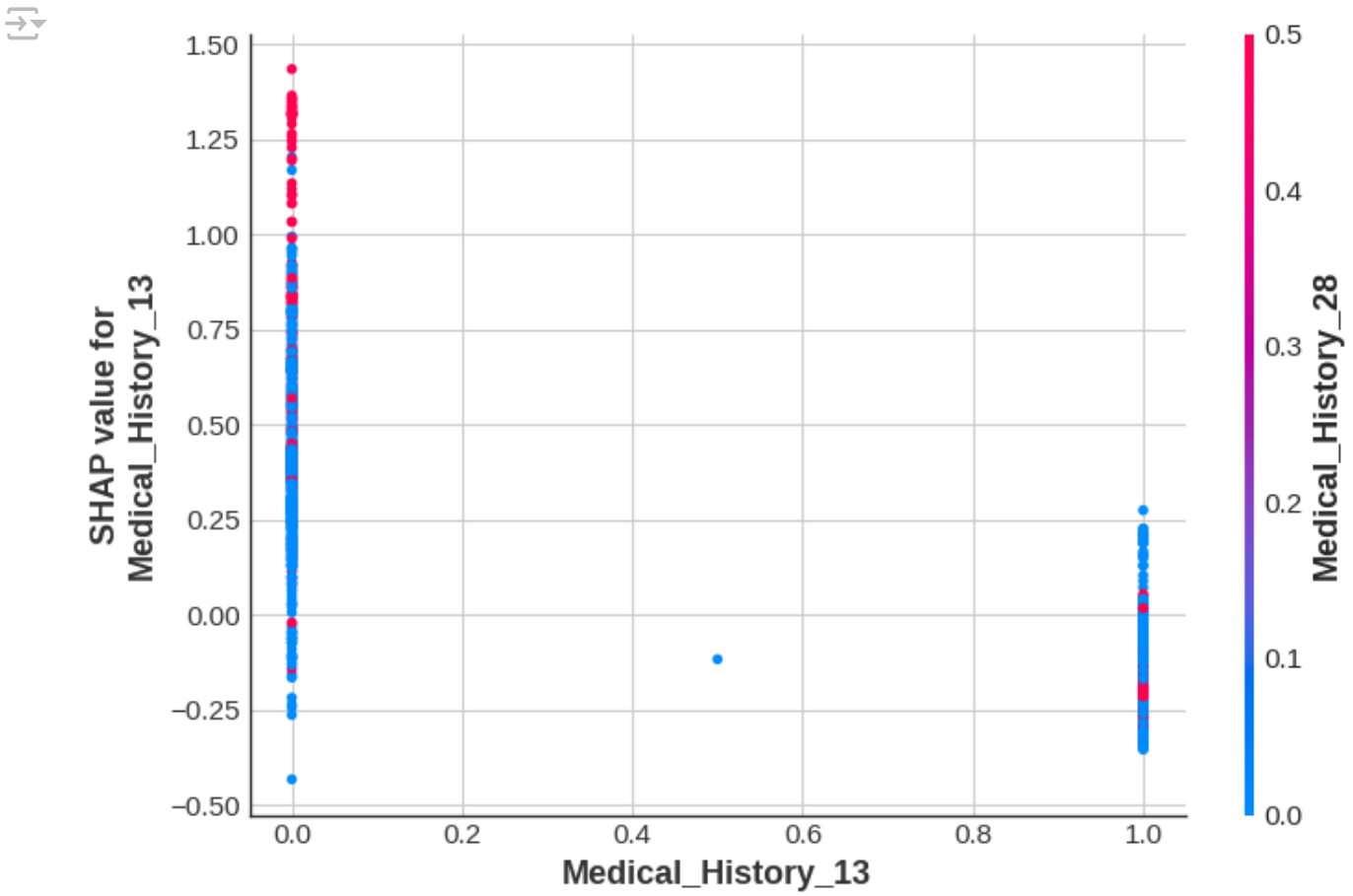
The color gradient represents Medical_History_13 values, from blue (low values) to red (high values), providing context on how it interacts with Medical_History_23.

Key observations include:

Binary Nature of Medical_History_23: The feature is binary, taking values of either 0 or 1. Medical_History_23 = 0: SHAP values are widely spead, indicating with Medical_13 influencing this effect.

Medical_History_23 = 1: SHAP values show a fairly wide range, suggesting a significant and variable influence on the risk classification. The color spread indicates that Medical_History_13 also plays a crucial role in modifying this impact.

```
shap.dependence_plot("rank(2)", shap_values[:, :, 0], X_test_L1reg)
```



Below SHAP dependence contribution plot to analyze the influence of BMI, as a function of Medical_History_16, on the model's risk classification predictions.

The horizontal axis represents BMI values from the dataset, while the vertical axis shows the SHAP values for BMI. Positive SHAP values indicate that the corresponding BMI values increase the likelihood of being classified into a certain risk category, whereas negative SHAP values suggest a decrease. A SHAP value near zero indicates that BMI has little to no impact on the classification.

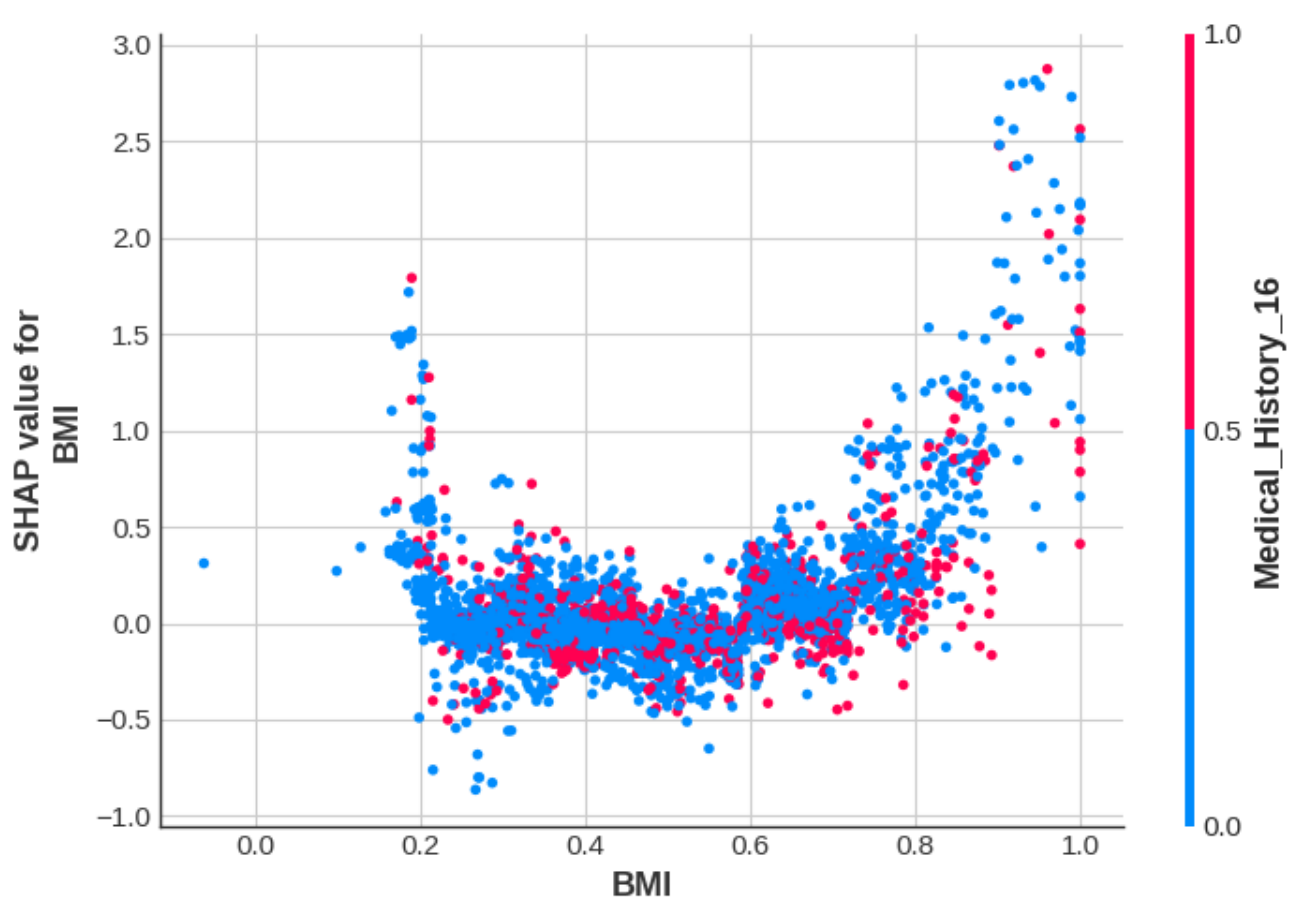
The color gradient represents the values of Medical_History_16, ranging from blue (low values) to red (high values). This additional dimension provides context on how Medical_History_16 interacts with BMI to influence risk classification.

Key observations include:

For very low BMI values (below 0.3), the SHAP values are highly variable, indicating that the impact of low BMI on risk classification is influenced by Medical_History_16. For mid-range BMI values (around 0.5), the SHAP values are generally close to zero, suggesting that BMI has a neutral effect on risk classification in this range. For very high BMI values (above 0.8), the SHAP values are predominantly positive, indicating that high BMI increases the likelihood of certain risk classifications. The spread of colors in this range suggests that Medical_History_16 significantly

interacts with BMI to modify the risk classification. This plot demonstrates how BMI influences the model's risk classification predictions, with Medical_History_16 providing critical context for understanding these interactions.

```
shap.dependence_plot("rank(3)", shap_values[:, :, 0], X_test_L1reg)
```



SHAP Dependence Plot (BMI vs Medical_History_16)

Here we generated a SHAP dependence contribution plot to analyze the influence of BMI, as a function of Medical_History_16, on the model's risk classification predictions.

The horizontal axis represents BMI values from the dataset, while the vertical axis shows the SHAP values for BMI. Positive SHAP values indicate that the corresponding BMI values increase the likelihood of being classified into a certain risk category, whereas negative SHAP values suggest a decrease. A SHAP value near zero indicates that BMI has little to no impact on the classification.

The color gradient represents the values of Medical_History_16, ranging from blue (low values) to red (high values). This additional dimension provides context on how Medical_History_16 interacts with BMI to influence risk classification.

Key observations include:

- For very low BMI values (below 0.3), the SHAP values are highly variable, indicating that the impact of low BMI on risk classification is influenced by Medical_History_16.
- For mid-range BMI values (around 0.5), the SHAP values are generally close to zero, suggesting that BMI has a neutral effect on risk classification in this range.
- For very high BMI values (above 0.8), the SHAP values are predominantly positive, indicating that high BMI increases the likelihood of certain risk classifications. The spread of colors in this range suggests that Medical_History_16 significantly interacts with BMI to modify the risk classification.

This plot demonstrates how BMI influences the model's risk classification predictions, with Medical_History_16 providing critical context for understanding these interactions.

Conclusion

Areas of Improvement¶

↳ 1 cell hidden

8. Unsupervised Methods

In this section we apply unsupervised methods to the dataset resulting from the Data Preparation section, which per below has 32, 62850 samples (life insurance applications) and 50 features (dimensions)

Here, as an initial visualisation we create a scatter plot taking a random 1000 samples from the data and using the first 10 feature dimensions, with colours corresponding to the 8 risk classifications.Although permitting an initial impression of the data, this approach is clearly insufficient for these data.

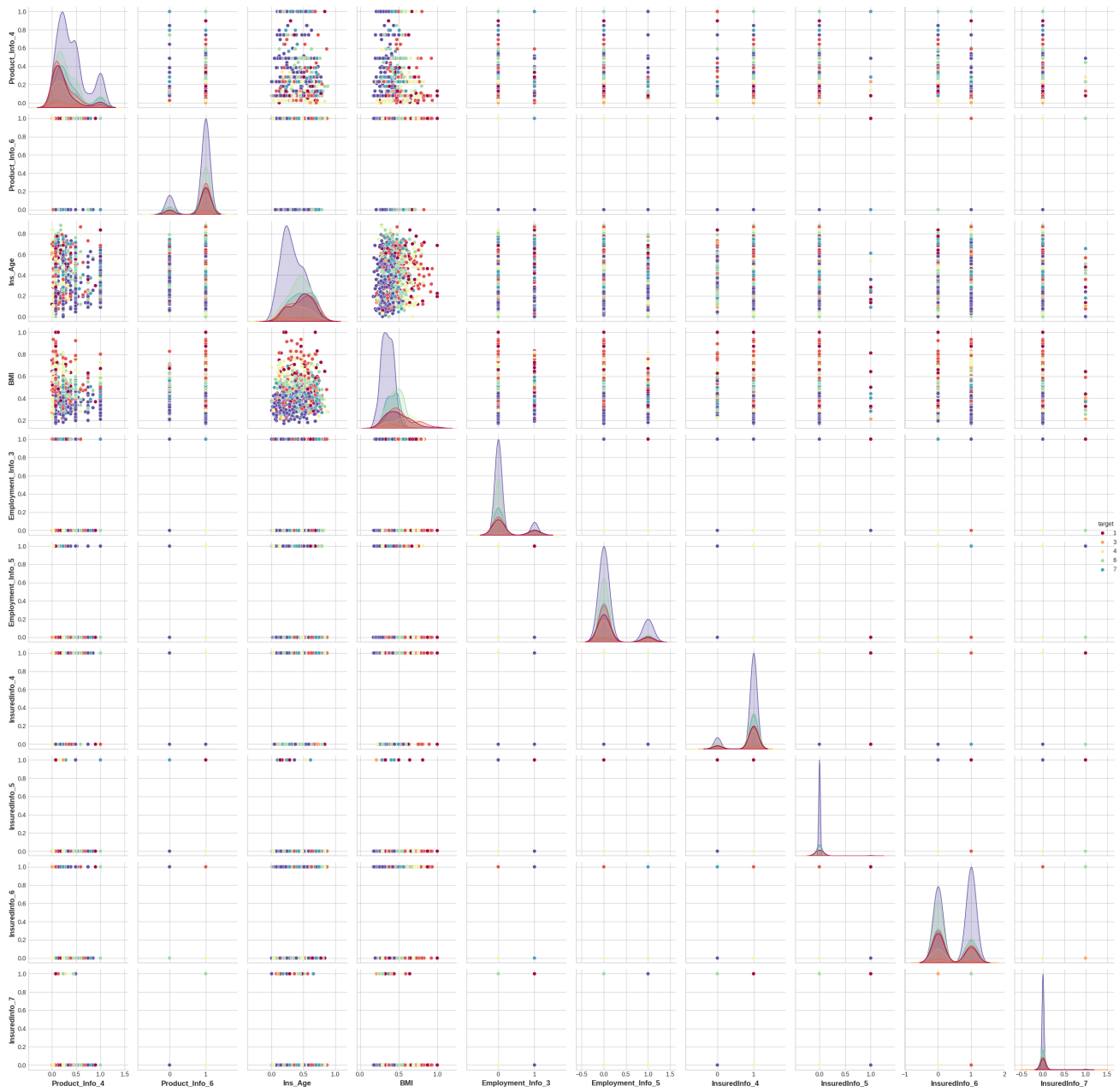
8.1 Pair Plots

```
X_train_L1reg.shape
```

```
(35628, 50)
```

```
%%time
# Sampling a subset of the data
sampled_data = X_train_L1reg.sample(n=1000, random_state=42)
# Add y to the sample df so we can plot the colours of the target variable
sampled_data['target'] = y_train[sampled_data.index]  # Assuming y_train is the ta
```

```
# Creating the pairplot for the first 10 dimensions with the sampled data
# sns.pairplot(sampled_data.iloc[:, :10], hue='target', palette='Spectral')
sns.pairplot(sampled_data.iloc[:, :10].join(sampled_data['target']), hue='target',
plt.show()
```




```
CPU times: user 1min 20s, sys: 816 ms, total: 1min 20s
Wall time: 1min 23s
```

8.2 Initial Visualisation - 2d

```
!pip install umap-learn
```

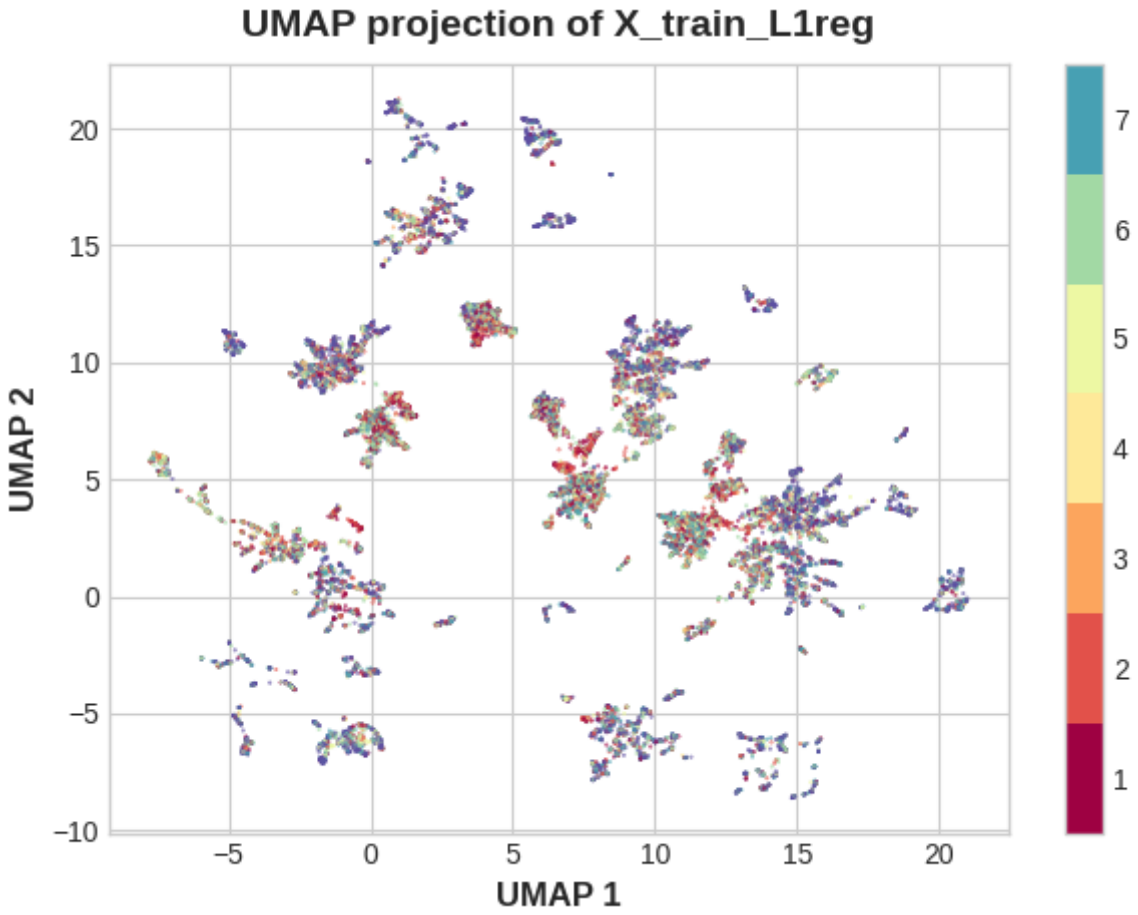
```
Collecting umap-learn
  Downloading umap_learn-0.5.6-py3-none-any.whl.metadata (21 kB)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: scipy>=1.3.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.10
Requirement already satisfied: numba>=0.51.2 in /usr/local/lib/python3.10/dist
Collecting pynndescent>=0.5 (from umap-learn)
  Downloading pynndescent-0.5.13-py3-none-any.whl.metadata (6.8 kB)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: llvmlite<0.44,>=0.43.0dev0 in /usr/local/lib/py
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Downloading umap_learn-0.5.6-py3-none-any.whl (85 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 85.7/85
Downloading pynndescent-0.5.13-py3-none-any.whl (56 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.9/56
Installing collected packages: pynndescent, umap-learn
Successfully installed pynndescent-0.5.13 umap-learn-0.5.6
```

```
import umap
```

Here, for initial visualisation, we reduce the data to 2-dimensions using UMAP. Ultimately, we will aim to cluster the data in high dimensional space and then visualise the result of that clustering. First, however, to help frame what follows, we simple view the data coloured by the risk category that each data point represents using a different colour for each classification.

```
# Assuming X_train_L1reg and y_train are already defined
standard_embedding = umap.UMAP(random_state=42).fit_transform(X_train_L1reg)

plt.scatter(standard_embedding[:, 0], standard_embedding[:, 1], c=y_train.astype(i
plt.colorbar(boundaries=np.arange(y_train.min(), y_train.max()+1)-0.5).set_ticks(r
plt.title('UMAP projection of X_train_L1reg')
plt.xlabel('UMAP 1')
plt.ylabel('UMAP 2')
plt.show()
```

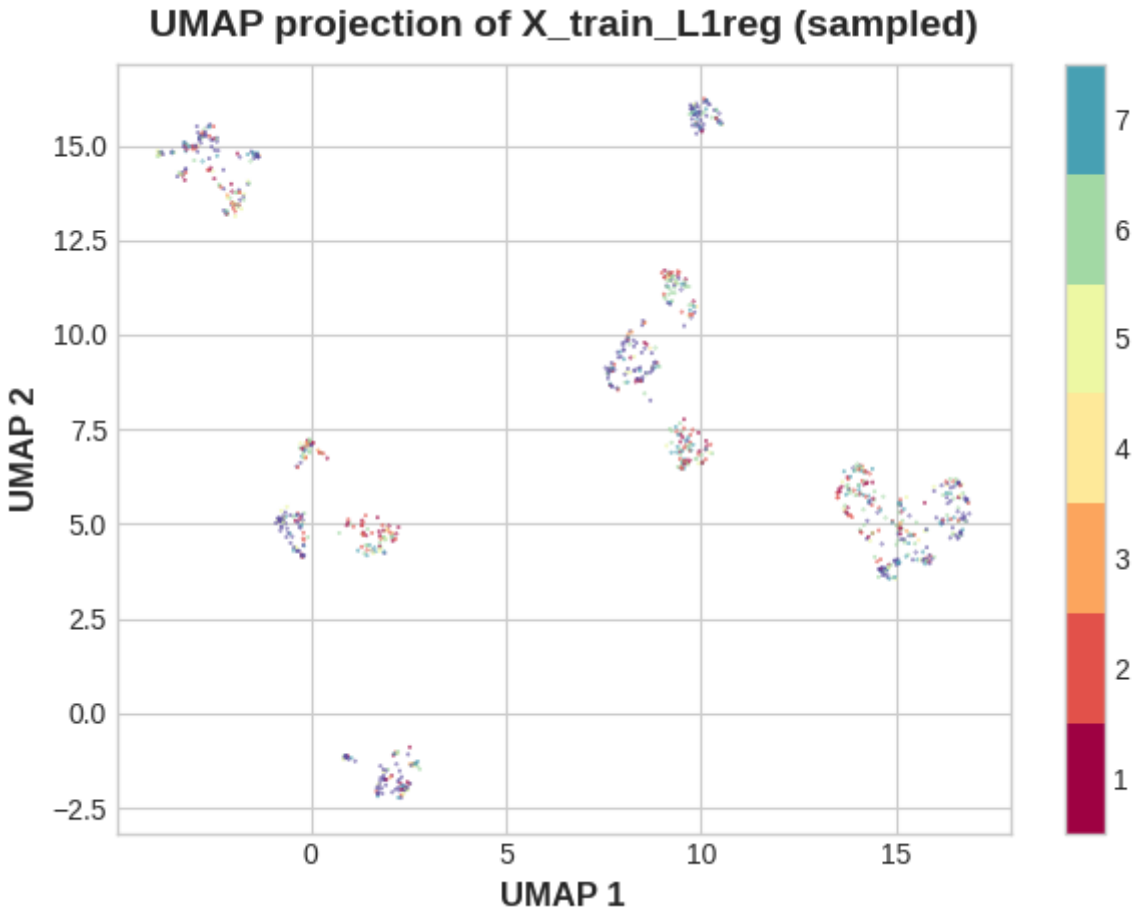
```
/usr/local/lib/python3.10/dist-packages/umap/umap_.py:1945: UserWarning: n_job
  warn(f"n_jobs value {self.n_jobs} overridden to 1 by setting random_state. U
```



Per above but sampling 1000 application records.

```
X_train_L1reg_sampled = X_train_L1reg.sample(n=1000, random_state=42)
# Add y to the sample df so we can plot the colours of the target variable
y_train_L1reg_sampled = y_train[X_train_L1reg_sampled.index] # Assuming y is the t


# Assuming X_train_L1reg and y_train are already defined
standard_embedding_sampled = umap.UMAP(random_state=42).fit_transform(X_train_L1re

plt.scatter(standard_embedding_sampled[:, 0], standard_embedding_sampled[:, 1], c=
plt.colorbar(boundaries=np.arange(y_train_sampled.min(), y_train_L1reg_sampled.max
plt.title('UMAP projection of X_train_L1reg (sampled)')
plt.xlabel('UMAP 1')
plt.ylabel('UMAP 2')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/umap/umap_.py:1945: UserWarning: n_job
  warn(f"n_jobs value {self.n_jobs} overridden to 1 by setting random_state. U
```



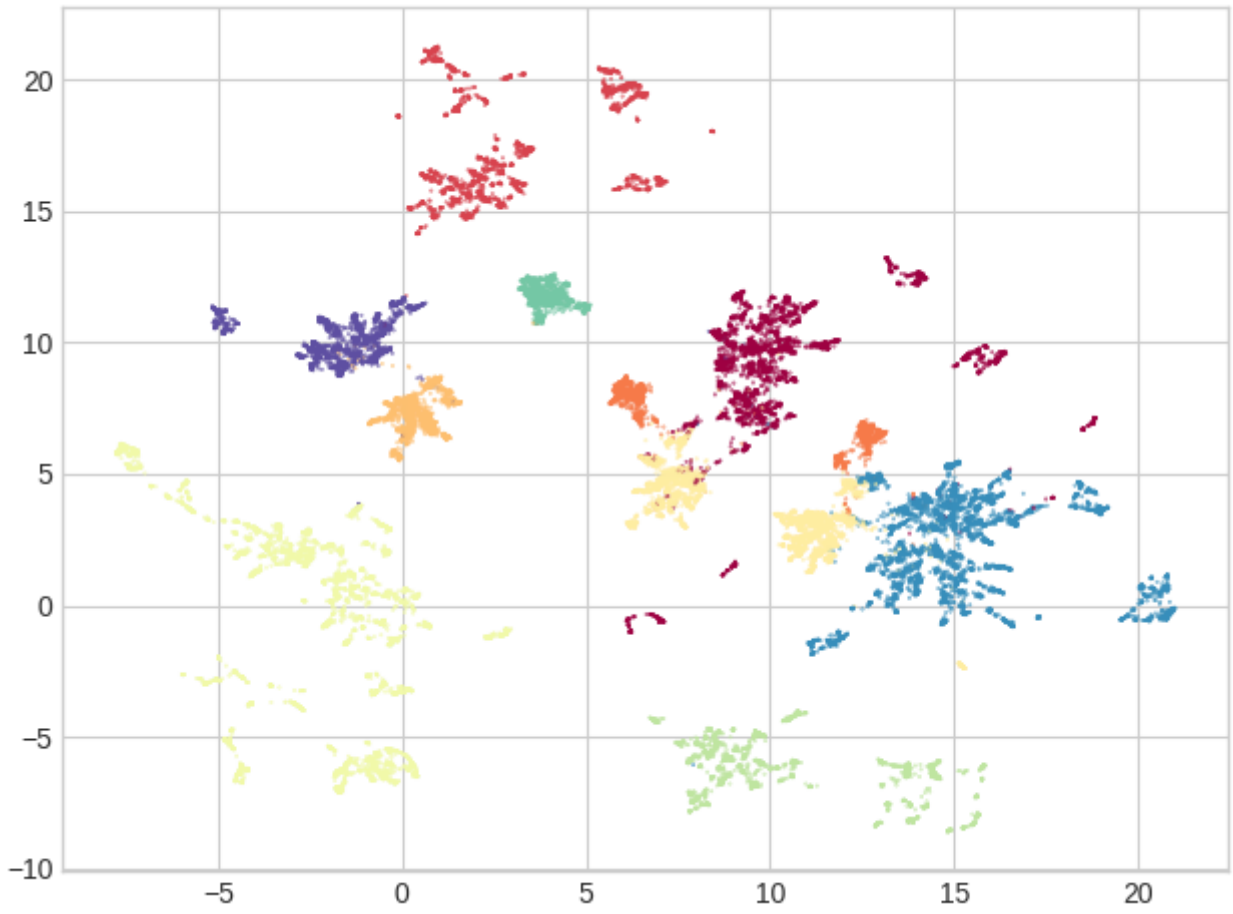
8.3 k-Means Clustering

Here, we cluster the data using traditional k-Means choosing k, the number of clusters to match the number of the target `Response` attribute risk categories (8). Thus, we provide the same number of clusters we are looking for.

```
import sklearn.cluster as cluster

kmeans_labels = cluster.KMeans(n_clusters=10).fit_predict(X_train_L1reg)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:1416: Futur
  super()._check_params_vs_input(X, default_n_init=10)
```

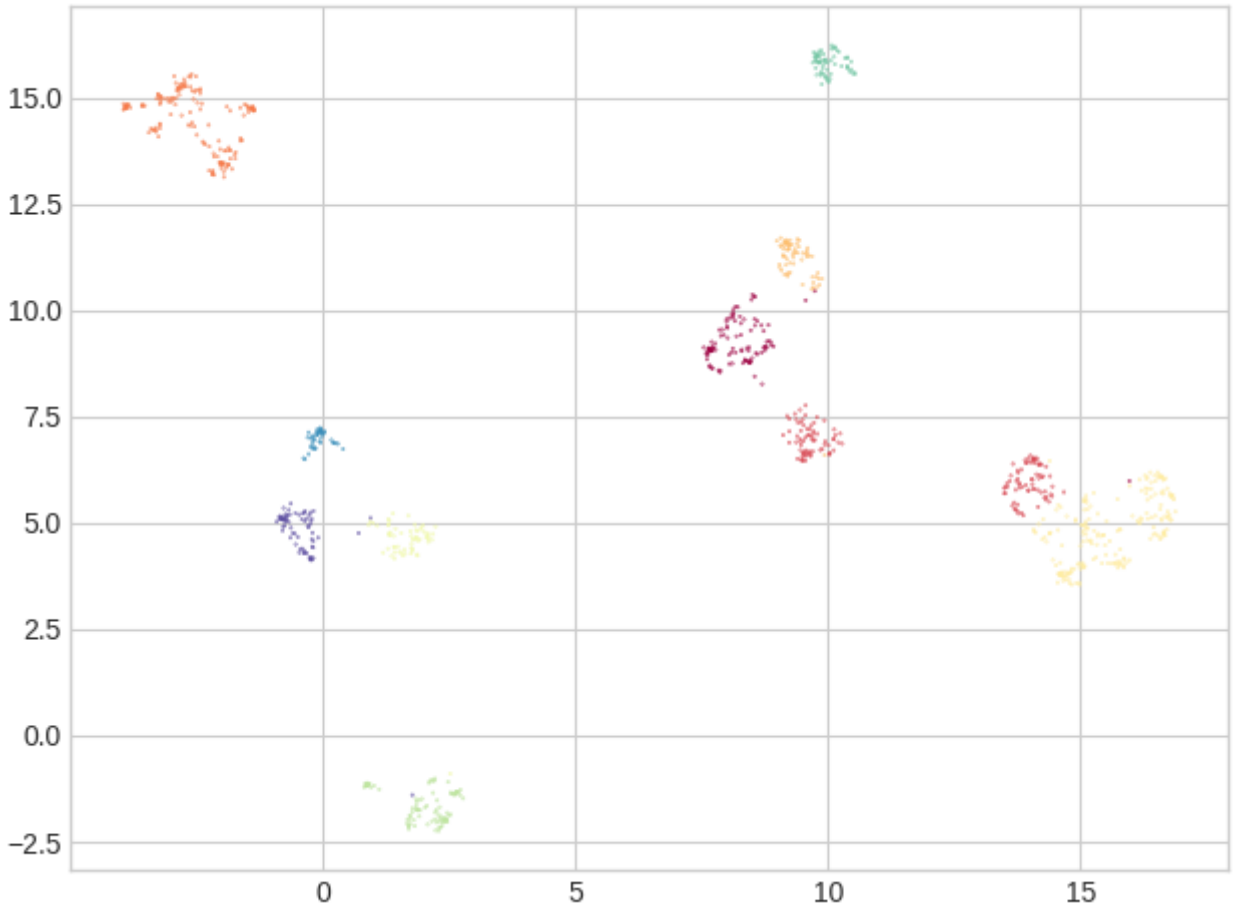
```
plt.scatter(standard_embedding[:, 0], standard_embedding[:, 1], c=kmeans_labels, s
```



The results show that the clusters are not tightly coalesced there is some potential as the colours are broadly associated with identifiable clusters in many cases.

```
kmeans_labels_sampled = cluster.KMeans(n_clusters=10).fit_predict(X_train_L1reg_sa
plt.scatter(standard_embedding_sampled[:, 0], standard_embedding_sampled[:, 1], c=
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:1416: Futur
  super()._check_params_vs_input(X, default_n_init=10)
```



The results show that the clusters are not tightly coalesced, there appears to be potential as the colours are generally associated with identifiable clusters in many cases. Multiple clusters are evident, a few dense or moderately dense, and clusters are well-separated. Fig. below shows the same k-Means drawn from 1000 samples and indeed shows the same.

```
# Verification check
print(y_train.shape)
print(X_train_L1reg.shape)
```

```
(35628,)
(35628, 50)
```

```
from sklearn.metrics import adjusted_rand_score, adjusted_mutual_info_score

# Assuming y_train_L1reg contains the true labels and kmeans_labels contains the l
ari = adjusted_rand_score(y_train, kmeans_labels)
ami = adjusted_mutual_info_score(y_train, kmeans_labels)

# Printing the results
print("Adjusted Rand Index (ARI):", ari)
print("Adjusted Mutual Information (AMI) Score:", ami)
```

```
Adjusted Rand Index (ARI): 0.03357865345117074
Adjusted Mutual Information (AMI) Score: 0.05105875394501763
```

Evaluating the k-means clustering results, using the Adjusted Rand Index (ARI) and Adjusted Mutual Information (AMI) Score, indicated:

1. **Adjusted Rand Index (ARI): 0.0336**
    - This score is quite low, suggesting that the clustering does not closely match the true class labels. It indicates poor agreement between the clusters and the ground truth.
2. **Adjusted Mutual Information (AMI) Score: 0.0511**
    - Similarly, this low AMI score indicates a weak mutual information between the clustering and the true class labels. It suggests that the clusters do not capture much of the information about the actual classes.

In summary, both the ARI and AMI scores suggest that the k-means clustering does not perform well in identifying the true underlying structure of the data. This suggests that k-means clustering does not perform well in identifying the true underlying structure of the data, so we will investigate more advanced clustering methods.

8.4 HDBSCAN

```
!pip install hdbscan
import hdbscan
```

```
Collecting hdbscan
  Downloading hdbscan-0.8.37-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x
Collecting cython<3,>=0.27 (from hdbscan)
  Downloading Cython-0.29.37-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x
Requirement already satisfied: numpy<2,>=1.20 in /usr/local/lib/python3.10/dis
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: scikit-learn>=0.20 in /usr/local/lib/python3.10
Requirement already satisfied: joblib>=1.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.
Downloading hdbscan-0.8.37-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.6/3.6
Downloading Cython-0.29.37-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9
Installing collected packages: cython, hdbscan
  Attempting uninstall: cython
    Found existing installation: Cython 3.0.10
    Uninstalling Cython-3.0.10:
      Successfully uninstalled Cython-3.0.10
Successfully installed cython-0.29.37 hdbscan-0.8.37
```

Before proceeding with HDBSCAN, for performance purposes, it can often be useful to reduce the dimensionality to around 50 or less, however, since we are already around this level through

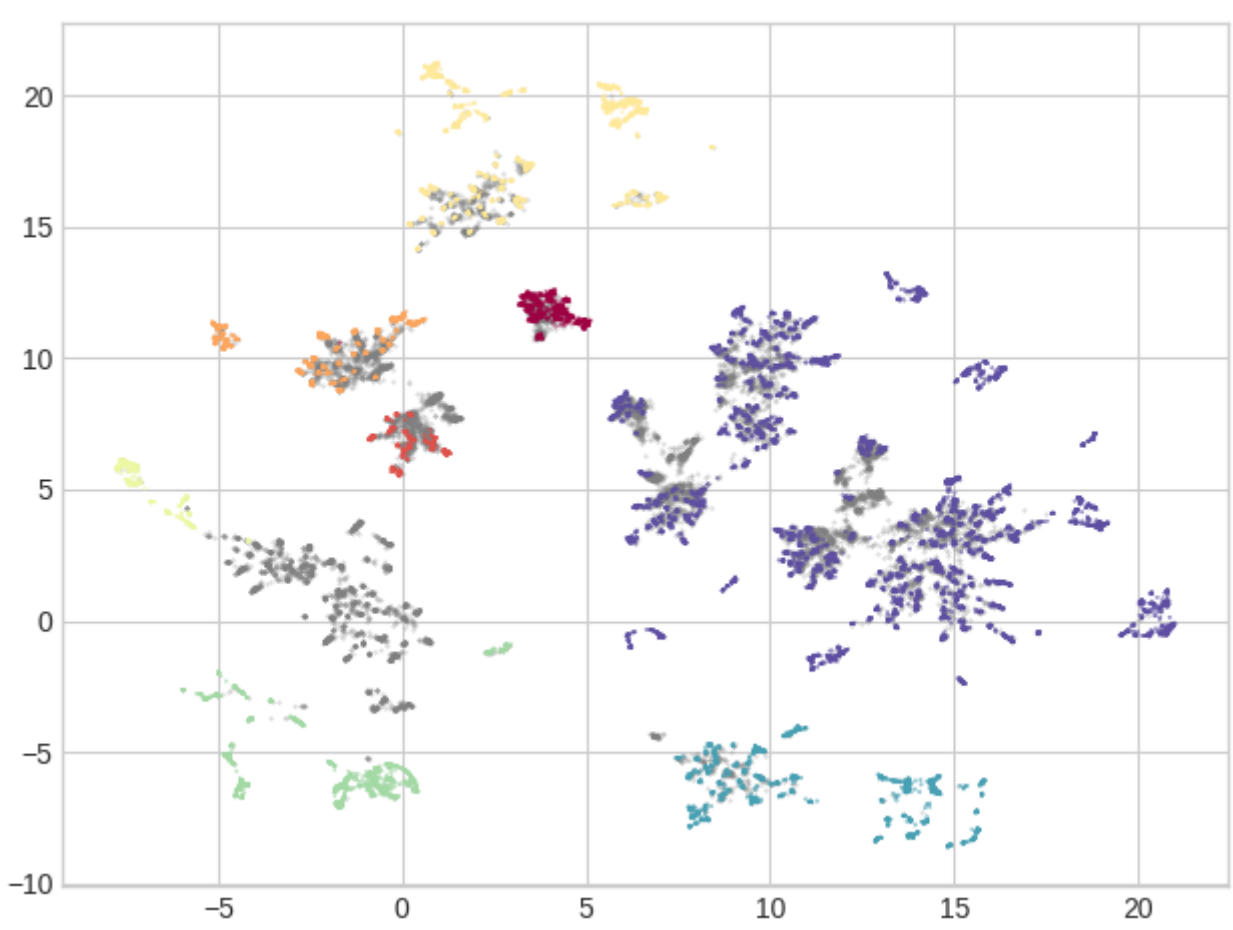
feature engineering alone we can omit this step.

One of the features of HDBSCAN is that it can refuse to cluster some points and classify them as “noise”. To visualize this aspect we will colour points that were classified as noise grey, and then colour the remaining points according to the cluster membership.

```
#lowd_mnist = PCA().fit_transform(mnisX_train_L1reg)
hdbscan_labels = hdbscan.HDBSCAN(min_samples=10, min_cluster_size=500).fit_predict
```

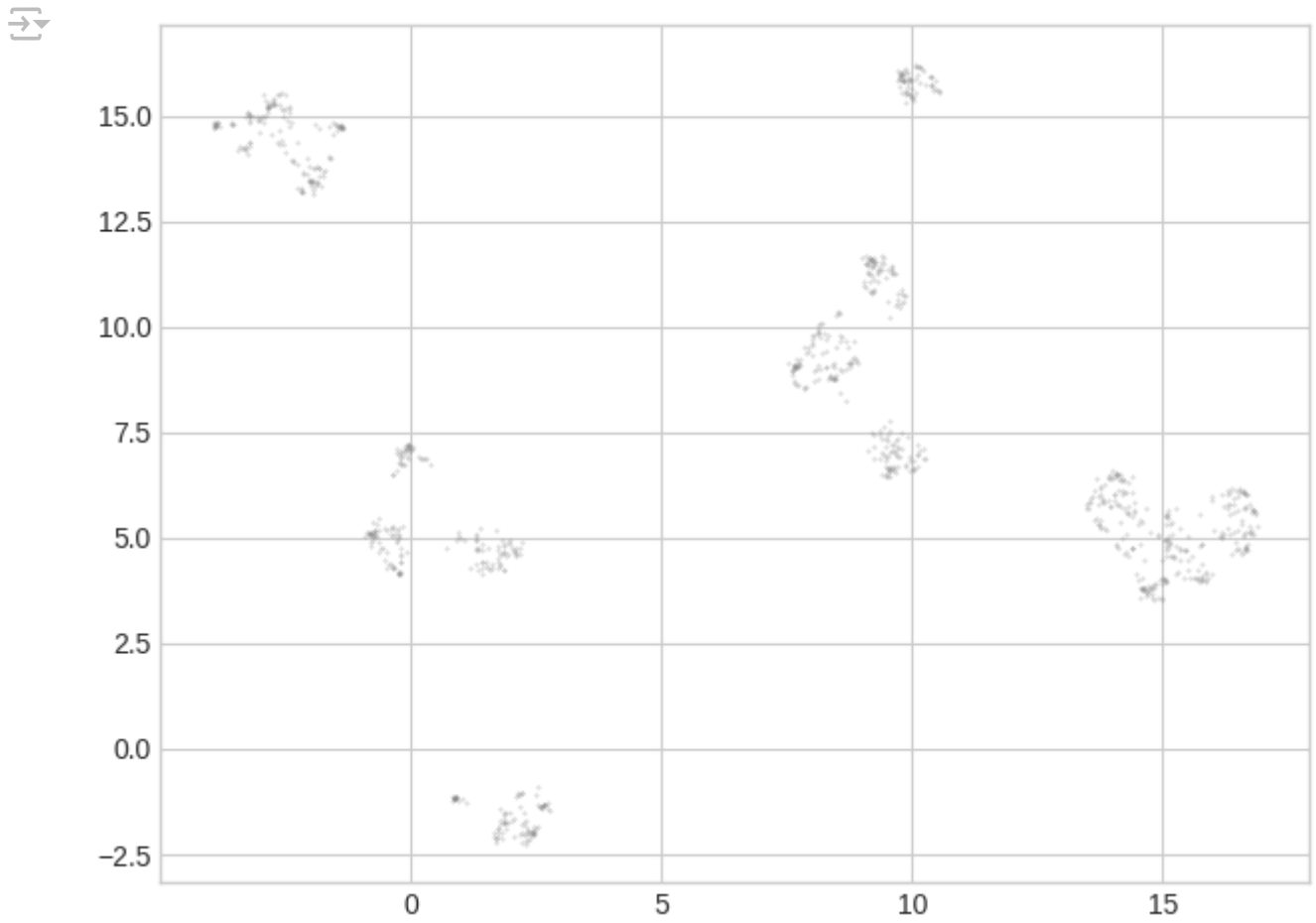
```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/backend/fork_exe
  pid = os.fork()
```

```
clustered = (hdbscan_labels >= 0)
plt.scatter(standard_embedding[~clustered, 0],
            standard_embedding[~clustered, 1],
            color=(0.5, 0.5, 0.5),
            s=0.1,
            alpha=0.5)
plt.scatter(standard_embedding[clustered, 0],
            standard_embedding[clustered, 1],
            cMB=hdbscan_labels[clustered],
            s=0.1,
            cmap='Spectral');
```




```
hdbscan_labels_sampled = hdbscan.HDBSCAN(min_samples=10, min_cluster_size=500).fit
```

```
clustered_sampled = (hdbscan_labels_sampled >= 0)
plt.scatter(standard_embedding_sampled[~clustered_sampled, 0],
            standard_embedding_sampled[~clustered_sampled, 1],
            color=(0.5, 0.5, 0.5),
            s=0.1,
            alpha=0.5)
plt.scatter(standard_embedding_sampled[clustered_sampled, 0],
            standard_embedding_sampled[clustered_sampled, 1],
            c=hdbscan_labels_sampled[clustered_sampled],
            s=0.1,
            cmap='Spectral');
```




```
# Assuming y_train_L1reg contains the true labels and kmeans_labels contains the l
ari = adjusted_rand_score(y_train, hdbscan_labels)
ami = adjusted_mutual_info_score(y_train, hdbscan_labels)

# Printing the results
print("Adjusted Rand Index (ARI):", ari)
print("Adjusted Mutual Information (AMI) Score:", ami)
```

```
Adjusted Rand Index (ARI): -0.013905530529543903
Adjusted Mutual Information (AMI) Score: 0.041957851216408465
```

```
clustered = (hdbscan_labels >= 0)

# Assuming y_train_L1reg contains the true labels and kmeans_labels contains the l
ari = adjusted_rand_score(y_train[clustered], hdbscan_labels[clustered])
ami = adjusted_mutual_info_score(y_train[clustered], hdbscan_labels[clustered])
```

```
# Printing the results
print("Adjusted Rand Index (ARI):", ari)
print("Adjusted Mutual Information (AMI) Score:", ami)
```

```
Adjusted Rand Index (ARI): -0.007738294010795304
Adjusted Mutual Information (AMI) Score: 0.03754900907033618
```

Similar to k-Means, the ARI and AMI score are very poor even when the cluster representing noise is omitted.

```
np.sum(clustered) / X_train_L1reg.shape[0]
```

```
0.5306781183338947
```

The challenge here is that HDBSCAN, as a density-based clustering algorithm, struggles with the curse of dimensionality; high-dimensional data require more samples to effectively represent density. By further reducing the dimensionality, we could enhance density visibility and facilitate more effective clustering by HDBSCAN. However, applying PCA for this reduction poses issues. Although PCA can reduce the 50 dimensions while preserving much of the data's variance, further reduction would significantly degrade performance due to PCA's linear nature. Therefore, more robust manifold learning is required according to \cite{sainburg2021parametric-umap}, which is where UMAP becomes beneficial.

8.5 UMAP

8.5.1 UMAP enhanced clustering (with HDBSCAN) - 2D

Our objective is to utilize UMAP to execute non-linear manifold-aware dimensionality reduction, enabling us to reduce the dataset to a number of dimensions low enough for a density-based clustering algorithm to function effectively. A key benefit of using UMAP is that it allows reduction to dimensions beyond just two – for instance, we can reduce to say 10 dimensions since the aim is clustering rather than visualisation, with UMAP imposing minimal performance overhead.

The challenge we have here is the complexity of the Prudential dataset and our requirement to ideally be able to reduce to two or three dimensions for purposes of visualisation and interaction by users (underwriters).

Here, we comment on setting parameters on the UMAP algorithm. When applying UMAP for dimensionality reduction, we need to choose different parameters than when using it for visualisation. Firstly, a higher \texttt{n_neighbors} value is advisable since smaller values will emphasise very local structures and are more likely to generate detailed cluster patterns that may be influenced by data noise rather than genuine clusters. For this purpose, we will double the default value from 15 to 30. Furthermore, it is favourable to set \texttt{min_dist} to a very low value. Because we aim to densely pack points together, a low value will facilitate this and also create clearer separations between clusters. Hence, we will set \texttt{min_dist} to 0.

```
clusterable_embedding_2d = umap.UMAP(
    n_neighbors=30,
    min_dist=0.0,
    n_components=2,
    random_state=42,
).fit_transform(X_train_L1reg)
```

```
/usr/local/lib/python3.10/dist-packages/umap/umap_.py:1945: UserWarning: n_job
  warn(f"n_jobs value {self.n_jobs} overridden to 1 by setting random_state. U
```

```
plt.scatter(clusterable_embedding_2d[:, 0], clusterable_embedding_2d[:, 1],
            c=y_train, s=0.1, cmap='Spectral')
```

```
<matplotlib.collections.PathCollection at 0x7e00ae91b850>
```