eda

October 5, 2023

[ ]:
```
# this file is a modified version of the original:
# https://www.kaggle.com/code/mateuszk013/
  ↪icr-eda-balanced-learning-with-lgbm-xgb
```

#

ICR - Identifying Age-Related Conditions

[1]:
```
import os
import shutil
import subprocess
from collections import defaultdict
from copy import copy
from functools import partial
from itertools import product
from pathlib import Path

# Sub-modules and so on.
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.figure_factory as ff
import plotly.graph_objects as go
import scipy.stats as stats

from colorama import Fore
from colorama import Style
from scipy.cluster.hierarchy import linkage
from scipy.spatial.distance import squareform
from scipy.stats import gaussian_kde
from scipy.stats import probplot
from IPython.core.display import HTML
from plotly.subplots import make_subplots

from xgboost import XGBClassifier
from lightgbm import LGBMClassifier

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

```
from sklearn.manifold import TSNE
from sklearn.model_selection import StratifiedKFold
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC
from sklearn.compose import make_column_selector
from sklearn.compose import make_column_transformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.feature_selection import f_classif
from sklearn.feature_selection import mutual_info_classif
from sklearn.impute import KNNImputer
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
from sklearn.metrics import brier_score_loss
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.preprocessing import Binarizer
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import StandardScaler

ON_KAGGLE = os.getenv("KAGGLE_KERNEL_RUN_TYPE") is not None

# Colorama settings.
CLR = (Style.BRIGHT + Fore.BLACK) if ON_KAGGLE else (Style.BRIGHT + Fore.WHITE)
RED = Style.BRIGHT + Fore.RED
BLUE = Style.BRIGHT + Fore.BLUE
CYAN = Style.BRIGHT + Fore.CYAN
RESET = Style.RESET_ALL

FONT_COLOR = "#010D36"
BACKGROUND_COLOR = "#F6F5F5"

CELL_HOVER = {  # for row hover use <tr> instead of <td>
    "selector": "td:hover",
    "props": "background-color: #F6F5F5",
}
TEXT_HIGHLIGHT = {
    "selector": "td",
    "props": "color: #FF2079; font-weight: bold",
}
INDEX_NAMES = {
```

```
    "selector": ".index_name",
    "props": "font-style: italic; background-color: #010D36; color: #F2F2F0;",
}
HEADERS = {
    "selector": "th:not(.index_name)",
    "props": "font-style: italic; background-color: #010D36; color: #F2F2F0;",
}
DF_STYLE = (INDEX_NAMES, HEADERS, TEXT_HIGHLIGHT)

# Utility functions.
def download_dataset_from_kaggle(user, dataset, directory):
    command = "kaggle datasets download -d "
    filepath = directory / (dataset + ".zip")

    if not filepath.is_file():
        subprocess.run((command + user + "/" + dataset).split())
        filepath.parent.mkdir(parents=True, exist_ok=True)
        shutil.unpack_archive(dataset + ".zip", "data")
        shutil.move(dataset + ".zip", "data")


def download_competition_from_kaggle(competition):
    command = "kaggle competitions download -c "
    filepath = Path("data/" + competition + ".zip")

    if not filepath.is_file():
        subprocess.run((command + competition).split())
        Path("data").mkdir(parents=True, exist_ok=True)
        shutil.unpack_archive(competition + ".zip", "data")
        shutil.move(competition + ".zip", "data")


# Html `code` block highlight.
HTML(
    """
<style>
code {
    background: rgba(58, 90, 129, 0.5) !important;
    border-radius: 4px !important;
    color: #f2f2f0 !important;
}
</style>
"""
)
```

[1]: <IPython.core.display.HTML object>

Competition Description

The goal of this competition is to predict if a person has any of three medical conditions. You are being asked to predict if the person has one or more of any of the three medical conditions (Class 1), or none of the three medical conditions (Class 0). You will create a model trained on measurements of health characteristics. To determine if someone has these medical conditions requires a long and intrusive process to collect information from patients. With predictive models, we can shorten this process and keep patient details private by collecting key characteristics relative to the conditions, then encoding these characteristics. Your work will help researchers discover the relationship between measurements of certain characteristics and potential patient conditions.

#

Quick Overview

Notes

Let's get started with a short dataset overview.

[2]:
```
competition = "icr-identify-age-related-conditions"

if not ON_KAGGLE:
    download_competition_from_kaggle(competition)
    train_path = "data/train.csv"
    test_path = "data/test.csv"
    greeks_path = "data/greeks.csv"
else:
    train_path = f"/kaggle/input/{competition}/train.csv"
    test_path = f"/kaggle/input/{competition}/test.csv"
    greeks_path = f"/kaggle/input/{competition}/greeks.csv"

train = pd.read_csv(train_path, index_col="Id").rename(columns=str.strip)
test = pd.read_csv(test_path, index_col="Id").rename(columns=str.strip)
greeks = pd.read_csv(greeks_path, index_col="Id").rename(columns=str.strip)
```

General Remarks

In the original description, we read that: The competition data comprises over fifty anonymized health characteristics linked to three age-related conditions. Your goal is to predict whether a subject has or has not been diagnosed with one of these conditions - a binary classification problem. Note that this is a Code Competition, in which the actual test set is hidden. In this version, we give some sample data in the correct format to help you author your solutions. When your submission is scored, this example test data will be replaced with the full test set. There are about 400 rows in the full test set.

Moreover, we know that:

train.csv - The training set.

Id - Unique identifier for each observation.

AB-GL - Fifty-six anonymized health characteristics. All are numeric except for EJ, which is categorical.

Class - A binary target: 1 indicates the subject has been diagnosed with one of the three conditions, 0 indicates they have not.

test.csv - The test set. Your goal is to predict the probability that a subject in this set belongs to each of the two classes.

greeks.csv - Supplemental metadata, only available for the training set.

Alpha - Identifies the type of age-related condition, if present.

A - No age-related condition. Corresponds to class 0.

B, D, G - The three age-related conditions. Correspond to class 1.

Beta, Gamma, Delta - Three experimental characteristics.

Epsilon - The date the data for this subject was collected. Note that all of the data in the test set was collected after the training set was collected.

[3]:
```
train.head().style.set_table_styles(DF_STYLE).format(precision=3)
```

[3]: <pandas.io.formats.style.Styler at 0x7b333355e080>

[4]:
```
train.info(verbose=False)
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 617 entries, 000ff2bfdfe9 to ffcca4ded3bb
Columns: 57 entries, AB to Class
dtypes: float64(55), int64(1), object(1)
memory usage: 279.6+ KB
```

[5]:
```
greeks.head().style.set_table_styles(DF_STYLE)
```

[5]: <pandas.io.formats.style.Styler at 0x7b333355df90>

[6]:
```
greeks.info(verbose=False)
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 617 entries, 000ff2bfdfe9 to ffcca4ded3bb
Columns: 5 entries, Alpha to Epsilon
dtypes: object(5)
memory usage: 28.9+ KB
```

[7]:
```
missing_values_cols = train.isna().sum()[train.isna().sum() > 0].index.to_list()

print(CLR + "Training Dataset Missing Values\n")

for feature in missing_values_cols:
    print(
        (CLR + feature) + "\t",
        (RED + str(train[feature].isna().sum())) + "\t",
```

```
        (RED + f"{train[feature].isna().sum() / len(train):.1%}" + RESET) +␣
↪"\t",
        (RED + f"{train[feature].dtype}"),
    )
```

```
Training Dataset Missing Values
BQ      60      9.7%        float64
CB      2       0.3%        float64
CC      3       0.5%        float64
DU      1       0.2%        float64
EL      60      9.7%        float64
FC      1       0.2%        float64
FL      1       0.2%        float64
FS      2       0.3%        float64
GL      1       0.2%        float64
```

[8]:
```
print(
    CLR + "Training Dataset Duplicated Rows:",
    RED + f"{train.drop('Class', axis=1).duplicated().sum()}",
)
```

```
Training Dataset Duplicated Rows: 0
```

[9]:
```
fig = px.pie(
    train.assign(ClassMap=train.Class.map({0: "Class 0", 1: "Class 1"})),
    names="ClassMap",
    height=540,
    width=840,
    hole=0.65,
    title="Target Overview - Class",
    color_discrete_sequence=["#010D36", "#FF2079"],
)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    showlegend=False,
)
fig.add_annotation(
    x=0.5,
    y=0.5,
    align="center",
```

```
    xref="paper",
    yref="paper",
    showarrow=False,
    font_size=22,
    text="Class<br>Imbalance",
)
fig.update_traces(
    hovertemplate=None,
    textposition="outside",
    texttemplate="%{label}<br>%{value} - %{percent}",
    textfont_size=16,
    rotation=-20,
    marker_line_width=25,
    marker_line_color=BACKGROUND_COLOR,
)
fig.show()
```

Observations

The training dataset is small, containing 617 samples. Nevertheless, we have to handle 57 different medical characteristics (attributes), including the binary target.

These features are anonymous, and we all know that these are specific medical characteristics.

We've got additional data, e.g. greeks.csv, but we will look at this later, especially the Epsilon attribute.

In our dataset, we have nine numeric features that contain missing values. Typically, only 1 to 3 values are missing for each attribute. However, there are two specific features where we observe 60 missing values each.

Lastly, there is quite a lot of unbalance in the target class: 83% (no age-related conditions) to 17% (at least one age-related condition).

#

Basic Relations in Numerical Features

Notes

Let's focus on an elementary description of numerical features. Firstly, let's see the numerical summary. Then, we will get to the correlation matrix and finally create hierarchical clustering based on Pearson correlations.

[10]:
```
numeric_descr = (
    train.drop("Class", axis=1)
    .describe(percentiles=[0.01, 0.05, 0.25, 0.50, 0.75, 0.95, 0.99])
    .drop("count")
    .T.rename(columns=str.title)
)

numeric_descr.style.set_table_styles(DF_STYLE).format(precision=3)
```

[10]: <pandas.io.formats.style.Styler at 0x7b333358bd60>

Observations

Let's look at the Q1-Q3 range and upper percentiles, including the max value. We may conclude that many of these distributions have long tails, which will probably require some transformations like log-level one.

[11]:
```
color_map = [[0.0, "#01CBEE"], [0.5, "#010D36"], [1.0, "#FF2079"]]

pearson_corr = (
    train.drop("Class", axis=1).corr(numeric_only=True, method="pearson").
 ↪round(2)
)
mask = np.triu(np.ones_like(pearson_corr, dtype=bool))
lower_triangular_corr = (
    pearson_corr.mask(mask)
    .dropna(axis="index", how="all")
    .dropna(axis="columns", how="all")
)

heatmap = go.Heatmap(
    z=lower_triangular_corr,
    x=lower_triangular_corr.columns,
    y=lower_triangular_corr.index,
    text=lower_triangular_corr.fillna(""),
    texttemplate="%{text}",
    xgap=1,
    ygap=1,
    showscale=True,
    colorscale=color_map,
    colorbar_len=1.02,
    hoverinfo="none",
)
fig = go.Figure(heatmap)
fig.update_layout(
    font_color=FONT_COLOR,
    title="Correlation Matrix (Pearson) - Lower Triangular",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    width=840,
    height=840,
    xaxis_showgrid=False,
    yaxis_showgrid=False,
    yaxis_autorange="reversed",
)
fig.show()
```

Observations

Here we have several highly correlated features like BZ vs BC (0.91) or DV vs CL (0.95). Such extreme linear correlations give hope for rejecting certain features.

[12]:
```
dissimilarity = 1 - np.abs(pearson_corr)

fig = ff.create_dendrogram(
    dissimilarity,
    labels=pearson_corr.columns,
    orientation="left",
    colorscale=px.colors.sequential.YlGnBu_r,
    # squareform() returns lower triangular in compressed form - as 1D array.
    linkagefun=lambda x: linkage(squareform(dissimilarity), method="complete"),
)
fig.update_layout(
    font_color=FONT_COLOR,
    title="Hierarchical Clustering using Correlation Matrix (Pearson)",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    height=1340,
    width=840,
    yaxis=dict(
        showline=False,
        title="Feature",
        ticks="",
    ),
    xaxis=dict(
        showline=False,
        title="Distance",
        ticks="",
        range=[-0.05, 1.05],
    ),
)
fig.update_traces(line_width=1.5)
fig.show()
```

Observations

As you can see, here we have minimal distances between BZ - BC, DV - CL, and EH - FD.

#

Pair Plots & Kernel Density Estimation

Notes

In this section, we will focus on exploring distributions in a general manner. Firstly, we will depict some pair plots of strongly correlated features, and then we will see the probability density of these variables by target value.

Let's define some small utility functions. The former is liable for KDE calculations, and the latter provides appropriate axes arrangement.

[13]:
```
def get_kde_estimation(data_series):
    kde = gaussian_kde(data_series.dropna())
    kde_range = np.linspace(
        data_series.min() - data_series.max() * 0.1,
        data_series.max() + data_series.max() * 0.1,
        len(data_series),
    )
    estimated_values = kde.evaluate(kde_range)
    estimated_values_cum = np.cumsum(estimated_values)
    estimated_values_cum /= estimated_values_cum.max()
    return kde_range, estimated_values, estimated_values_cum


def get_n_rows_axes(n_features, n_cols=5, n_rows=None):
    n_rows = int(np.ceil(n_features / n_cols))
    current_col = range(1, n_cols + 1)
    current_row = range(1, n_rows + 1)
    return n_rows, list(product(current_row, current_col))
```

[14]:
```
corr_threshold = 0.7

highest_abs_corr = (
    lower_triangular_corr.abs()
    .unstack()
    .sort_values(ascending=False)  # type: ignore
    .rename("Absolute Pearson Correlation")
)

highest_abs_corr = (
    highest_abs_corr[highest_abs_corr > corr_threshold]
    .to_frame()
    .reset_index(names=["Feature 1", "Feature 2"])
)

highest_corr_combinations = highest_abs_corr[["Feature 1", "Feature 2"]].
 ↪to_numpy()
highest_abs_corr.style.set_table_styles(DF_STYLE).format(precision=2)
```

[14]:
```
<pandas.io.formats.style.Styler at 0x7b33335c79d0>
```

[15]:
```
n_cols = 3
n_rows, axes = get_n_rows_axes(len(highest_corr_combinations), n_cols=n_cols)

fig = make_subplots(
```

```
    rows=n_rows,
    cols=n_cols,
    horizontal_spacing=0.1,
    vertical_spacing=0.06,
)

show_legend = True

for k, ((current_row, current_col), (feature1, feature2)) in enumerate(
    zip(axes, highest_corr_combinations)
):
    if k > 0:
        show_legend = False

    fig.add_scatter(
        x=train.query("Class == 0")[feature1],
        y=train.query("Class == 0")[feature2],
        mode="markers",
        name="Class 0",
        marker=dict(color="#010D36", size=3, symbol="diamond", opacity=0.5),
        legendgroup="Class 0",
        showlegend=show_legend,
        row=current_row,
        col=current_col,
    )
    fig.add_scatter(
        x=train.query("Class == 1")[feature1],
        y=train.query("Class == 1")[feature2],
        mode="markers",
        name="Class 1",
        marker=dict(color="#FF2079", size=2, symbol="circle", opacity=0.5),
        legendgroup="Class 1",
        showlegend=show_legend,
        row=current_row,
        col=current_col,
    )
    fig.update_xaxes(
        type="log",
        title_text=feature1,
        titlefont_size=9,
        titlefont_family="Arial Black",
        tickfont_size=7,
        row=current_row,
        col=current_col,
    )
    fig.update_yaxes(
        type="log",
```

```
        title_text=feature2,
        titlefont_size=9,
        titlefont_family="Arial Black",
        tickfont_size=7,
        row=current_row,
        col=current_col,
    )

fig.update_annotations(font_size=14)
fig.update_layout(
    font_color=FONT_COLOR,
    title="Highest Pearson Correlations - Pair Plots<br>Double Logarithmic␣
 ↪Scale",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    width=840,
    height=1140,
    legend=dict(
        orientation="h",
        yanchor="bottom",
        xanchor="right",
        y=1.01,
        x=1,
        itemsizing="constant",
    ),
)

fig.show()
```

Observations

In the case of this dataset, it's impossible to show all pair-plots, so I chose only those most correlated.

The highest correlation is between EH - FD (0.97), and this is clearly visible there. Moreover, values associated with Class 0 are shifted towards higher values. You can explore this by turning off and turning on a given group using legend. A similar situation occurs within DU - EH and DU - FD. Unfortunately, we don't know what these abbreviations mean.

Moreover, we can see that many different values of a given feature correspond to one specific value from the second one. It may account for a little problem for machine learning algorithms. Such a situation appears in each of the above relationships.

[16]:
```
numeric_data = train.select_dtypes("number")
numeric_cols = numeric_data.drop("Class", axis=1).columns.tolist()

n_cols = 5
n_rows, axes = get_n_rows_axes(len(numeric_cols))
```

```
fig1 = make_subplots(
    rows=n_rows,
    cols=n_cols,
    y_title="Probability Density",
    horizontal_spacing=0.06,
    vertical_spacing=0.04,
)
fig2 = copy(fig1)

show_legend = True

for k, ((current_row, current_col), feature) in enumerate(zip(axes,␣
 ↪numeric_cols)):
    if k > 0:
        show_legend = False

    for target, color in zip((0, 1), ("#010D36", "#FF2079")):
        kde_range, estimated_values, estimated_values_cum = get_kde_estimation(
            numeric_data.query(f"Class == {target}")[feature]
        )

        for fig, kde_values in zip(  # type: ignore
            (fig1, fig2), (estimated_values, estimated_values_cum)
        ):
            fig.add_scatter(
                x=kde_range,
                y=kde_values,
                line=dict(dash="solid", color=color, width=1),
                fill="tozeroy",
                name=f"Class {target}",
                legendgroup=f"Class {target}",
                showlegend=show_legend,
                row=current_row,
                col=current_col,
            )
            fig.update_yaxes(
                tickfont_size=7,
                row=current_row,
                col=current_col,
            )
            fig.update_xaxes(
                title_text=feature,
                titlefont_size=9,
                titlefont_family="Arial Black",
                tickfont_size=7,
                row=current_row,
                col=current_col,
```

```
            )

title1 = "Numerical Features - Kernel Density Estimation"
title2 = "Numerical Features - Cumulative Kernel Density Estimation"

for fig, title in zip((fig1, fig2), (title1, title2)):
    fig.update_annotations(font_size=14)
    fig.update_layout(
        font_color=FONT_COLOR,
        title=title,
        title_font_size=18,
        plot_bgcolor=BACKGROUND_COLOR,
        paper_bgcolor=BACKGROUND_COLOR,
        width=840,
        height=1340,
        legend=dict(
            orientation="h",
            yanchor="bottom",
            xanchor="right",
            y=1.01,
            x=1,
        ),
    )

fig1.show()
```

Observations

You can activate and deactivate distributions for a certain class by clicking on the legend.

Well, here we've got a diversity of variables, i.e. some of them probably relatively good fit a normal distribution (BN, CU, GH), some have long tails (and extremely long tails), like AR, AY, BR, BZ, etc. Moreover, there are even bimodal distributions (CW, EL and GL).

We will better understand the diversity between classes on the cumulative plots, as below.

[17]:
```
fig2.show()
```

Observations

The cumulative KDE reveals a varied presence of long tails in the given distributions. Depending on the variable, the responsibility for the long tail can be attributed to values associated with Class 0 in some cases, while in other cases it is associated with values linked to Class 1. Additionally, there are instances where the distributions overlap.

#

Probability Plots & Transformations

Notes

This section aims to explore so-called probability plots. It's a pleasant graphical technique to assess whether a variable follows a specific distribution. Here, the normal one. On such a plot, samples which follow normal distribution are deployed on a diagonal straight line.

Some machine learning models assume that the variable follows a normal distribution. In turn, the mentioned technique helps to decide which transformations should be done within the given variable to improve the fit to that distribution.

Let's get started with original values and see results.

[18]:
```
fig = make_subplots(
    rows=n_rows,
    cols=n_cols,
    y_title="Observed Values",
    x_title="Theoretical Quantiles",
    horizontal_spacing=0.06,
    vertical_spacing=0.04,
)
fig.update_annotations(font_size=14)

for (row, col), feature in zip(axes, numeric_cols):
    (osm, osr), (slope, intercept, R) = probplot(train[feature].dropna(),␣
 ↪rvalue=True)
    x_theory = np.array([osm[0], osm[-1]])
    y_theory = intercept + slope * x_theory
    R2 = f"R\u00b2 = {R * R:.2f}"
    fig.add_scatter(x=osm, y=osr, mode="markers", row=row, col=col,␣
 ↪name=feature)
    fig.add_scatter(x=x_theory, y=y_theory, mode="lines", row=row, col=col)
    fig.add_annotation(
        x=-1.25,
        y=osr[-1] * 0.75,
        text=R2,
        showarrow=False,
        row=row,
        col=col,
        font_size=9,
    )
    fig.update_yaxes(tickfont_size=7, row=row, col=col)
    fig.update_xaxes(
        title_text=feature,
        titlefont_size=9,
        titlefont_family="Arial Black",
        tickfont_size=7,
        row=row,
        col=col,
    )
```

```
fig.update_layout(
    font_color=FONT_COLOR,
    title="Numerical Features - Probability Plots against Normal Distribution",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    showlegend=False,
    width=840,
    height=1340,
)
fig.update_traces(
    marker=dict(size=1, symbol="x-thin", line=dict(width=2, color="#010D36")),
    line_color="#FF2079",
)
fig.show()
```

Observations

As you can see, some variables fit a normal distribution well, which manifests by a high coefficient of determination (R-squared) and evenly deployed samples around the straight line. These are for example DN or BN.

Nevertheless, there are a lot of features which do not fit the normal one. We can improve that by specific transformations:

Log Transformation - generally works fine with right-skewed data. Requires non-negative numbers.

Square Root Transformation - similarly to log-level transformation. Requires non-negative numbers.

Square Transformation - helps to reduce left-skewed data.

Reciprocal Transformation - used sometimes, when data is skewed, or there are obvious outliers. Not defined at zero.

Box-Cox Transformation - used when data is skewed or has outliers. Requires strictly positive numbers.

Yeo-Johnson Transformation - variation of Box-Cox transformation, but without restrictions concerning numbers.

Let's check all of these transformations for our variables. We simply use the probplot() function to get R-squared coefficients for each transformation.

[19]:
```
r2_scores = defaultdict(tuple)

for feature in numeric_cols:
    orig = train[feature].dropna()
    _, (*_, R_orig) = probplot(orig, rvalue=True)
    _, (*_, R_log) = probplot(np.log(orig), rvalue=True)
    _, (*_, R_sqrt) = probplot(np.sqrt(orig), rvalue=True)
    _, (*_, R_reci) = probplot(np.reciprocal(orig), rvalue=True)
    _, (*_, R_boxcox) = probplot(stats.boxcox(orig)[0], rvalue=True)
```

```
    _, (*_, R_yeojohn) = probplot(stats.yeojohnson(orig)[0], rvalue=True)
    r2_scores[feature] = (
        R_orig * R_orig,
        R_log * R_log,
        R_sqrt * R_sqrt,
        R_reci * R_reci,
        R_boxcox * R_boxcox,
        R_yeojohn * R_yeojohn,
    )

r2_scores = pd.DataFrame(
    r2_scores, index=("Original", "Log", "Sqrt", "Reciprocal", "BoxCox",␣
 ↪"YeoJohnson")
).T

r2_scores["Winner"] = r2_scores.idxmax(axis=1)
r2_scores.style.set_table_styles(DF_STYLE).format(precision=3)
```

[19]:
```
<pandas.io.formats.style.Styler at 0x7b3332961ae0>
```

[20]:
```
no_transform_cols = r2_scores.query("Winner == 'Original'").index
log_transform_cols = r2_scores.query("Winner == 'Log'").index
sqrt_transform_cols = r2_scores.query("Winner == 'Sqrt'").index
reciprocal_transform_cols = r2_scores.query("Winner == 'Reciprocal'").index
boxcox_transform_cols = r2_scores.query("Winner == 'BoxCox'").index
yeojohnson_transform_cols = r2_scores.query("Winner == 'YeoJohnson'").index
```

[21]:
```
AB_orig = train.AB.dropna()
(osm, osr), (slope, intercept, R) = probplot(AB_orig, rvalue=True)
x_theory = np.array([osm[0], osm[-1]])
y_theory = intercept + slope * x_theory

fig = make_subplots(
    rows=1,
    cols=2,
    subplot_titles=["Probability Plot against Normal Distribution",␣
 ↪"Histogram"],
)

fig.add_scatter(x=osm, y=osr, mode="markers", row=1, col=1, name="AB")
fig.add_scatter(x=x_theory, y=y_theory, mode="lines", row=1, col=1)
fig.add_annotation(
    x=-1.25,
    y=osr[-1] * 0.4,
    text=f"R\u00b2 = {R * R:.3f}",
    showarrow=False,
    row=1,
```

```
    col=1,
)
fig.update_yaxes(title_text="Observed Values", row=1, col=1)
fig.update_xaxes(title_text="Theoretical Quantiles", row=1, col=1)
fig.update_traces(
    marker=dict(size=1, symbol="x-thin", line=dict(width=2, color="#010D36")),
    line_color="#FF2079",
)

fig.add_histogram(
    x=AB_orig,
    marker_color="#010D36",
    opacity=0.75,
    name="AB",
    row=1,
    col=2,
)
fig.update_yaxes(title_text="Count", row=1, col=2)
fig.update_xaxes(title_text="AB", row=1, col=2)

fig.update_layout(
    font_color=FONT_COLOR,
    title="AB Feature - Original",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    showlegend=False,
    width=840,
    height=440,
    bargap=0.2,
)

fig.update_annotations(font_size=14)
fig.show()
```

[22]:
```
AB_transformed = stats.boxcox(train.AB.dropna())[0]
(osm, osr), (slope, intercept, R) = probplot(AB_transformed, rvalue=True)
x_theory = np.array([osm[0], osm[-1]])
y_theory = intercept + slope * x_theory

fig = make_subplots(
    rows=1,
    cols=2,
    subplot_titles=["Probability Plot against Normal Distribution",␣
 ↪"Histogram"],
)
```

```
fig.add_scatter(x=osm, y=osr, mode="markers", row=1, col=1, name="BoxCox(AB)")
fig.add_scatter(x=x_theory, y=y_theory, mode="lines", row=1, col=1)
fig.add_annotation(
    x=-1.25,
    y=osr[-1] * 0.4,
    text=f"R\u00b2 = {R * R:.3f}",
    showarrow=False,
    row=1,
    col=1,
)
fig.update_yaxes(title_text="Observed Values", row=1, col=1)
fig.update_xaxes(title_text="Theoretical Quantiles", row=1, col=1)
fig.update_traces(
    marker=dict(size=1, symbol="x-thin", line=dict(width=2, color="#010D36")),
    line_color="#FF2079",
)

fig.add_histogram(
    x=AB_transformed,
    marker_color="#010D36",
    opacity=0.75,
    name="BoxCox(AB)",
    row=1,
    col=2,
)
fig.update_yaxes(title_text="Count", row=1, col=2)
fig.update_xaxes(title_text="BoxCox(AB)", row=1, col=2)

fig.update_layout(
    font_color=FONT_COLOR,
    title="AB Feature - Box-Cox Transformation",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    showlegend=False,
    width=840,
    height=440,
    bargap=0.2,
)

fig.update_annotations(font_size=14)
fig.show()
```

Observations

As you can see above, the Box-Cox transformation works perfectly for the AB variable.

Obviously, I suppose we will be working with tree-based models at the end, but sometimes models

like SVC handle very well, and appropriate transformations for these algorithms are crucial.

Let's look closely at these values we've got.

[23]:
```
r2_scores.describe().T.drop("count", axis=1).rename(
    columns=str.title
).style.set_table_styles(DF_STYLE).format(precision=3)
```

[23]: <pandas.io.formats.style.Styler at 0x7b3330b2ae60>

Observations

Well, as you can see Yeo-Johnson's transformation wins in most cases. However, some simple transformations, like log one, are also doing well. Moreover, we have one feature where none of the transformations helps - CW.

#

Semi-Constant Features

Notes

There is something suspect with some variables, i.e., these that have especially poor transformation results. Let's specify, and look at them closely: AR, AY, BZ, DF, and DV.

[24]:
```
problematic_variables = train[["AR", "AY", "BZ", "DF", "DV"]]
problematic_variables.head(10).style.set_table_styles(DF_STYLE)
```

[24]: <pandas.io.formats.style.Styler at 0x7b33325582e0>

[25]:
```
problematic_variables.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 617 entries, 000ff2bfdfe9 to ffcca4ded3bb
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   AR      617 non-null    float64
 1   AY      617 non-null    float64
 2   BZ      617 non-null    float64
 3   DF      617 non-null    float64
 4   DV      617 non-null    float64
dtypes: float64(5)
memory usage: 28.9+ KB
```

[26]:
```
problematic_variables_descr = numeric_descr.loc[problematic_variables.columns]
problematic_variables_descr.style.set_table_styles(DF_STYLE).format(precision=3)
```

[26]: <pandas.io.formats.style.Styler at 0x7b3332cef1c0>

[27]:
```
problematic_variables_vs_class = problematic_variables.join(train.Class)
duplicated_rows = problematic_variables_vs_class.duplicated(
    subset=["AR", "AY", "BZ", "DF", "DV"]
)

duplicates = problematic_variables_vs_class[duplicated_rows]
no_duplicates = problematic_variables_vs_class[~duplicated_rows]

print(
    CLR
    + "Ratio of duplicated / not duplicated rows in ['AR', 'AY', 'BZ', 'DF',␣
 ↪'DV'] subset:\n"
)
print(CLR + "Duplicated rows:".ljust(20), RED + f"{len(duplicates)}")
print(CLR + "Not duplicated rows:".ljust(20), RED + f"{len(no_duplicates)}\n")

print(CLR + "Class balance when ['AR', 'AY', 'BZ', 'DF', 'DV'] are duplicated:
 ↪\n")
for key, value in duplicates.Class.value_counts(normalize=True).to_dict().
 ↪items():
    print(CLR + f"Class {key}:", RED + f"{value:.1%}")

print(CLR + "\nClass balance when ['AR', 'AY', 'BZ', 'DF', 'DV'] are not␣
 ↪duplicated:\n")
for key, value in no_duplicates.Class.value_counts(normalize=True).to_dict().
 ↪items():
    print(CLR + f"Class {key}:", RED + f"{value:.1%}")
```

```
Ratio of duplicated / not duplicated rows in ['AR', 'AY', 'BZ', 'DF',
'DV'] subset:
Duplicated rows:     280
Not duplicated rows: 337
Class balance when ['AR', 'AY', 'BZ', 'DF', 'DV'] are duplicated:
Class 0: 91.8%
Class 1: 8.2%


Class balance when ['AR', 'AY', 'BZ', 'DF', 'DV'] are not duplicated:
Class 0: 74.8%
Class 1: 25.2%
```

Observations

Okay, there we have just mostly the same value for the whole variable, additionally with significant

outliers. That's the reason for weak transformations. Moreover, we know that some of them (AR, BZ and DV) strongly correlate with some other features, but it's hard to say about correlation since almost the whole distribution consists of one value. Nevertheless, we could probably drop these attributes.

Remains the question about AY and DF. These ones do not have such strong correlations with any features. The AY correlates with EP (0.52), and DF with AR (0.35) and EU (0.30).

It's good to check other semi-constant variables. In such a case, we probably should binarize them. Let's suppose we consider semi-constant features where the minimum value and median are the same.

[28]:
```
semi_constant_mask = np.isclose(numeric_descr["Min"], numeric_descr["50%"])
semi_constant_descr = numeric_descr[semi_constant_mask]
semi_constant_descr.style.set_table_styles(DF_STYLE).format(precision=3)
```

[28]: <pandas.io.formats.style.Styler at 0x7b33335cd2a0>

[29]:
```
semi_constant_features_corr = (
    train[np.r_[semi_constant_descr.index, ["Class"]]]
    .corr(method="pearson")["Class"]
    .to_dict()
)

print(CLR + "Semi-constant features - Pearson correlation with Class:\n")
for feature, corr_with_class in semi_constant_features_corr.items():
    print((CLR + feature + ":") + "\t", (RED + f"{corr_with_class:+.3f}"))
```

```
Semi-constant features - Pearson correlation with Class:

AH:     +0.045
AR:     +0.064
AY:     +0.082
BC:     +0.156
BZ:     +0.112
CL:     +0.017
DF:     +0.064
DV:     +0.015
EP:     -0.068
GE:     -0.071
Class:  +1.000
```

[30]:
```
# Let's save these features with their median thresholds.
semi_const_cols_thresholds = semi_constant_descr["50%"].to_dict()
```

Observations

Weak correlations with Class give hope that binarization should not be harmful. What is more, perhaps these features ought to be dropped.

#

EJ - The Only One Categorical Variable

Notes

In the whole dataset, there is only one categorical feature - EJ. Let's focus on this.

[31]:
```
sunburst_df = train.copy()
sunburst_df.Class = sunburst_df.Class.map({0: "Class 0", 1: "Class 1"})
sunburst_df.EJ = sunburst_df.EJ.map({"A": "EJ - A", "B": "EJ - B"})

fig = px.sunburst(
    sunburst_df,
    title="Class (Binary Target) vs EJ (Categorical)",
    path=["EJ", "Class"],
    color_discrete_sequence=["#010D36", "#FF2079"],
    height=640,
    width=640,
)
fig.update_traces(
    insidetextorientation="horizontal",
    texttemplate="%{label}<br>%{value} - %{percentParent}",
    marker_line_width=5,
    marker_line_color=BACKGROUND_COLOR,
)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
)
fig.show()
```

[32]:
```
EJ_pivot = (
    train.pivot_table(
        values="Class",
        index="EJ",
        aggfunc=["mean", "sum", "count"],
        margins=True,
        margins_name="Total",
    )
    .rename(
        columns={
            "mean": "Class 1 Fraction",
```

```
            "sum": "Class 1 Count",
            "count": "Samples",
        }
    )
    .droplevel(level=1, axis="columns")
)

EJ_pivot.style.set_table_styles(DF_STYLE)
```

[32]: <pandas.io.formats.style.Styler at 0x7b333299cd60>

Observations

I think there is nothing suspect.

#

Dimensionality Reduction with t-SNE

Notes

The t-SNE algorithm is an excellent tool to reduce data dimensionality and visualize datasets. Additionally, it tries to group together similar samples. We will use it with the given dataset to see whether there are some clusters or something interesting in 2D and 3D.

Firstly, we provide simple preprocessing and transformations, which we explored in the previous section about probability plots. So far, I do not include binarization.

[33]:
```
casual_preprocess = make_pipeline(
    make_column_transformer(
        (
            StandardScaler(),
            no_transform_cols.to_list(),
        ),
        (
            make_pipeline(
                FunctionTransformer(func=np.log,␣
 ↪feature_names_out="one-to-one"),
                StandardScaler(),
            ),
            log_transform_cols.to_list(),
        ),
        (
            make_pipeline(
                FunctionTransformer(func=np.reciprocal,␣
 ↪feature_names_out="one-to-one"),
                StandardScaler(),
            ),
            reciprocal_transform_cols.to_list(),
        ),
```

```
        (
            PowerTransformer(method="box-cox", standardize=True),
            boxcox_transform_cols.to_list(),
        ),
        (
            PowerTransformer(method="yeo-johnson", standardize=True),
            yeojohnson_transform_cols.to_list(),
        ),
        (
            make_pipeline(
                SimpleImputer(strategy="most_frequent"),
                OrdinalEncoder(handle_unknown="use_encoded_value",␣
 ↪unknown_value=-1),
            ),
            make_column_selector(dtype_include=object),  # type: ignore
        ),
        remainder="drop",
        verbose_feature_names_out=False,
    ),
    KNNImputer(n_neighbors=10, weights="distance"),
)
```

[34]:
```
X = train.drop("Class", axis=1)
y = train.Class

X_processed = casual_preprocess.fit_transform(X)
X_processed_frame = pd.DataFrame(
    X_processed,
    columns=casual_preprocess.get_feature_names_out(),
    index=X.index,
)
X_processed_frame.head().style.set_table_styles(DF_STYLE).format(precision=3)
```

[34]:
```
<pandas.io.formats.style.Styler at 0x7b33335c7f10>
```

[35]:
```
tsne_2D = TSNE(n_components=2, n_jobs=-1, random_state=42, perplexity=10)
tsne_3D = TSNE(n_components=3, n_jobs=-1, random_state=42, perplexity=10)

X_2D = pd.DataFrame(
    tsne_2D.fit_transform(X_processed), columns=["dim1", "dim2"], index=X.index
).join(y.astype(str))

X_3D = pd.DataFrame(
    tsne_3D.fit_transform(X_processed), columns=["dim1", "dim2", "dim3"],␣
 ↪index=X.index
).join(y.astype(str))
```

[36]:
```
fig = px.scatter(
    X_2D.reset_index(),
    x="dim1",
    y="dim2",
    symbol="Class",
    symbol_sequence=["diamond", "circle"],
    color="Class",
    color_discrete_sequence=["#010D36", "#FF2079"],
    category_orders={"Class": ("0", "1")},
    hover_data="Id",
    opacity=0.6,
    height=840,
    width=840,
    title="Training Dataset - 2D Projection with t-SNE",
)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    legend=dict(
        orientation="h",
        yanchor="bottom",
        xanchor="right",
        y=1.05,
        x=1,
        title="Class",
        itemsizing="constant",
    ),
)
fig.update_traces(marker_size=6)
fig.show()
```

[37]:
```
fig = px.scatter_3d(
    X_3D.reset_index(),
    x="dim1",
    y="dim2",
    z="dim3",
    symbol="Class",
    symbol_sequence=["diamond", "circle"],
    color="Class",
    color_discrete_sequence=["#010D36", "#FF2079"],
    category_orders={"Class": ("0", "1")},
    hover_data="Id",
    opacity=0.6,
    height=840,
    width=840,
```

```
    title="Training Dataset - 3D Projection with t-SNE",
)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    legend=dict(
        orientation="h",
        yanchor="bottom",
        xanchor="right",
        y=1.05,
        x=1,
        title="Class",
        itemsizing="constant",
    ),
)
fig.update_traces(marker_size=3)
fig.show()
```

Observations

Well, as you can see, there is actually something interesting. I mean that cluster which is associated with Class 1. This behaviour occurs both in the 2D projection and 3D one. Moreover, I checked several different perplexity values and different random seeds. However, in each case, there is a smaller or bigger cluster. So probably, thereby, hangs a tale.

On the other hand, many samples overlap.

#

Feature Importance & Permutation Tests

Notes

In this section, we will tackle the general feature importance problem. We check several different methods to assess which variables are essential in the decision process.

Firstly, let's take completely different methods: LinearDiscriminantAnalysis, LGBMClassifier and mutual_info_classif(), and see what we get.

[38]:
```
lda_pipeline = make_pipeline(
    casual_preprocess,
    LinearDiscriminantAnalysis(),
).fit(X, y)
lda_info = np.abs(lda_pipeline[-1].scalings_.ravel())
lda_info = lda_info / lda_info.sum()  # Normalise to 1 to compare with other␣
 ↪methods.

lgbm_pipeline = make_pipeline(
    casual_preprocess,
```

```
    LGBMClassifier(random_state=42, is_unbalance=True),
).fit(X, y)
lgbm_info = lgbm_pipeline[-1].feature_importances_
lgbm_info = lgbm_info / lgbm_info.sum()

mutual_info = mutual_info_classif(
    X=casual_preprocess.fit_transform(X), y=y, random_state=42
)
mutual_info = mutual_info / np.sum(mutual_info)

importances = pd.DataFrame(
    [lda_info, lgbm_info, mutual_info],
    columns=lda_pipeline[0].get_feature_names_out(),
    index=["LDA", "LGBM", "MI"],
).T

importances[:10].style.set_table_styles(DF_STYLE).format(precision=4)
```

[38]: <pandas.io.formats.style.Styler at 0x7b33309ff280>

[39]:
```
importances_melted_frame = (
    importances.melt(
        var_name="Method",
        value_name="Importance",
        ignore_index=False,
    )
    .reset_index()
    .rename(columns={"index": "Feature"})
    .round(4)
)

fig = px.bar(
    importances_melted_frame,
    x="Importance",
    y="Feature",
    color="Importance",
    facet_col="Method",
    facet_col_spacing=0.07,
    height=940,
    width=840,
    color_continuous_scale=color_map,
    title="Normalised Feature Importances (Three Different Default Methods)",
)
fig.update_annotations(font_size=14)
fig.update_yaxes(
    matches=None,
    showticklabels=True,
```

```
    categoryorder="total ascending",
    tickfont_size=8,
)
fig.update_xaxes(matches=None)
fig.update_traces(width=0.7)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    coloraxis_colorbar=dict(
        orientation="h",
        title_side="bottom",
        yanchor="bottom",
        xanchor="center",
        title=None,
        y=-0.2,
        x=0.5,
    ),
)
fig.show()
```

Observations

So, as you can see, different methods give different results. However, these obtained from LGBM and Mutual Information are similar. In turn, LDA gave utterly different outcome. In that method, the feature importance measure is based on discriminator weights.

Okay, so the first sight is that the DU variable occupies a very high place in each method (in two of them, it wins). Moreover, GL, which won in the LDA method, is also high in the rest. What is more interesting, the LDA says that EJ (the only one categorical variable) is the fifth most important feature in the dataset. Meanwhile, LGBM says that it's useless. In the Mutual Information method, EJ is around in the middle.

Probably not all variables will be needed in the final model. We can explore this with a more sophisticated method based on out-of-bag data. We will perform the so-called permutation test to see when the balanced log loss metric is mostly sensitive while permuting samples in a certain feature.

[40]:
```
def balanced_log_loss(y_true, y_pred, **kwargs):
    """Competition evaluation metric - balanced logarithmic loss.
    The overall effect is such that each class is roughly equally
    important for the final score."""
    N0, N1 = np.bincount(y_true)

    y0 = np.where(y_true == 0, 1, 0)
    y1 = np.where(y_true == 1, 1, 0)

    eps = kwargs.get("eps", 1e-15)
```

```
    y_pred = np.clip(y_pred, eps, 1 - eps)
    p0 = np.log(1 - y_pred)
    p1 = np.log(y_pred)

    return -(1 / N0 * np.sum(y0 * p0) + 1 / N1 * np.sum(y1 * p1)) * 0.5
```

[41]:
```
n_bags = 10
n_folds = 5

np.random.seed(42)
seeds = np.random.randint(0, 19937, size=n_bags)
```

[42]:
```
original_loglosses = []
permutation_loglosses = pd.DataFrame()

forest = RandomForestClassifier(
    class_weight="balanced", criterion="log_loss", random_state=42
)
svc = SVC(class_weight="balanced", probability=True, random_state=42)
lgbm = LGBMClassifier(is_unbalance=True, random_state=42)

for classifier in (forest, svc, lgbm):
    y_proba_original = np.zeros_like(y, dtype=np.float64)
    y_proba_shuffled = defaultdict(partial(np.zeros_like, y, dtype=np.float64))

    for seed in seeds:
        skfold = StratifiedKFold(n_splits=n_folds, shuffle=True,␣
 ↪random_state=seed)
        classifier.set_params(random_state=seed)

        for train_ids, valid_ids in skfold.split(X, y):
            X_train, y_train = X.iloc[train_ids], y.iloc[train_ids]
            X_valid, y_valid = X.iloc[valid_ids], y.iloc[valid_ids]

            X_train = casual_preprocess.fit_transform(X_train)
            X_valid = casual_preprocess.transform(X_valid)

            classifier.fit(X_train, y_train)
            y_proba_original[valid_ids] += classifier.predict_proba(X_valid)[:,␣
 ↪1]

            for i, feature in enumerate(casual_preprocess.
 ↪get_feature_names_out()):
                X_shuffled = X_valid.copy()
                X_shuffled[:, i] = np.random.permutation(X_shuffled[:, i])  #␣
 ↪type: ignore
```

```
                y_proba_shuffled[feature][valid_ids] += classifier.
↪predict_proba(
                    X_shuffled
                )[:, 1]

    classifier_name = classifier.__class__.__name__
    feature_names = y_proba_shuffled.keys()

    original_loglosses.append(balanced_log_loss(y, y_proba_original / n_bags))
    permutation_loglosses[classifier_name] = pd.Series(
        [
            balanced_log_loss(y, y_proba_shuffled[feature] / n_bags)
            for feature in feature_names
        ],
        index=feature_names,
    )
```

Observations

The provided code requires clarification. Firstly, we will explore how rearranging samples within a specific feature affects the balanced logarithmic loss when evaluating the validation dataset. We begin with three distinct classifiers, and each of them is trained and evaluated using stratified cross-validation. The model is trained on a subset of the data and assessed on a separate subset during each cross-validation iteration. Consequently, we gather predictions for the entire dataset. To ensure more reliable outcomes, this entire process is repeated ten times with different random seeds, and the final outcome is averaged. Ultimately, we compute the balanced logarithmic loss. Importantly, throughout this entire process, we shuffle samples in the chosen feature of the validation subset and record results obtained from evaluating this modified dataset in a separate dictionary. If the variable is significant, we should observe worsened results in terms of balanced log loss. If the feature is really relevant, rather each classifier should show that.

[43]:
```
permutation_results_melted = (
    permutation_loglosses.melt(
        var_name="Method",
        value_name="Balanced Log Loss",
        ignore_index=False,
    )
    .reset_index()
    .rename(columns={"index": "Feature"})
    .round(4)
)

fig = px.bar(
    permutation_results_melted,
    x="Balanced Log Loss",
    y="Feature",
    color="Balanced Log Loss",
    facet_col="Method",
```

```
    facet_col_spacing=0.07,
    height=940,
    width=840,
    color_continuous_scale=color_map,
    title="Permutation Test Results - Balanced Log Loss when Permuting␣
 ↪Samples<br>"
    "in Certain Features (Averaged over Stratified 5-Fold and 10 Different␣
 ↪Seeds)",
)
fig.update_annotations(font_size=14)
fig.update_traces(width=0.7)
fig.update_xaxes(matches=None)
fig.update_yaxes(
    matches=None,
    showticklabels=True,
    categoryorder="total ascending",
    tickfont_size=8,
)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    coloraxis_colorbar=dict(
        orientation="h",
        title_side="bottom",
        yanchor="bottom",
        xanchor="center",
        title=None,
        y=-0.2,
        x=0.5,
    ),
    margin_t=120,
)
for original_logloss, max_logloss, col in zip(
    original_loglosses, permutation_loglosses.max().tolist(), (1, 2, 3)
):
    fig.add_vline(
        x=original_logloss,
        line_width=2,
        line_dash="dash",
        line_color="#FF2079",
        col=col,
    )
    fig.add_vrect(
        x0=original_logloss,
        x1=max_logloss,
```

```
        line_width=0,
        fillcolor="#FF2079",
        opacity=0.2,
        col=col,
    )
fig.show()
```

Observations

Each of the models says the same. The DU variable has the highest influence on predictions. Also important are, for example, AB or BQ.

Given the above facts, we probably should provide some feature selection steps in the final pipeline. There are a lot of methods to select features, so we need to explore them.

#

Look at Greeks

Notes

In this section, we will have a quick look at greeks metadata.

Let's get started with a parallel coordinates plot. Since greeks are categorical in total, we will check which categories are connected with each other.

[44]:
```
greeks = greeks.join(train.Class)
greeks_cats = greeks[["Alpha", "Beta", "Gamma", "Delta"]].astype("category")
greeks_codes = greeks_cats.apply(lambda x: x.cat.codes)
```

[45]:
```
fig = go.Figure(
    go.Parcoords(
        dimensions=[
            dict(
                label="Beta",
                values=greeks_codes.Beta,
                tickvals=np.unique(greeks_codes.Beta),
                ticktext=greeks_cats.Beta.cat.categories,
            ),
            dict(
                label="Gamma",
                values=greeks_codes.Gamma,
                tickvals=np.unique(greeks_codes.Gamma),
                ticktext=greeks_cats.Gamma.cat.categories,
            ),
            dict(
                label="Delta",
                values=greeks_codes.Delta,
                tickvals=np.unique(greeks_codes.Delta),
                ticktext=greeks_cats.Delta.cat.categories,
            ),
```

```
            dict(
                label="Alpha",
                values=greeks_codes.Alpha,
                tickvals=np.unique(greeks_codes.Alpha),
                ticktext=greeks_cats.Alpha.cat.categories,
            ),
            dict(
                label="Class",
                values=greeks.Class,
                tickvals=np.unique(greeks.Class),
            ),
        ],
        line=dict(
            color=greeks.Class,
            colorscale=color_map,
            showscale=True,
            colorbar=dict(
                title="Class",
                orientation="h",
                title_side="bottom",
                yanchor="bottom",
                xanchor="center",
                y=-0.35,
                x=0.5,
                nticks=2,
            ),
        ),
    )
)

fig.update_layout(
    font_color=FONT_COLOR,
    title="Greeks - Parallel Coordinates",
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
    height=540,
    width=840,
)
fig.update_traces(
    labelfont=dict(family="Arial Black", size=10),
    tickfont=dict(family="Arial Black", size=10),
    selector=dict(type="parcoords"),
)
fig.show()
```

Observations

Okay, so as you can see, the category "A" in the Beta characteristic is associated with people who have got age-related conditions. The rest, i.e. "B" and "C" are mixed. A more interesting situation occurs in the Gamma variable. There we have eight categories; six of them are related to age-related conditions, and two are not. Moving forward to the Delta feature, we see the situation is mixed in all categories.

Let's see the pivot table for these features. We will see Beta and Delta vs Class since the situation is diverse there.

[46]:
```
pivot = (
    greeks.pivot_table(
        values="Class",
        index=["Beta", "Delta"],
        aggfunc=["mean", "sum", "count"],
        margins=True,
        margins_name="Total",
    )
    .rename(
        columns={
            "mean": "Class 1 Fraction",
            "sum": "Class 1 Count",
            "count": "Samples",
        }
    )
    .droplevel(level=1, axis="columns")
)

pivot.style.set_table_styles(DF_STYLE)
```

[46]:
```
<pandas.io.formats.style.Styler at 0x7b3332559750>
```

Observations

So, we have only eight people with Beta - "A" and Delta - "A" indicators, and all of them had age-related conditions. Moreover, there is quite a big group with Beta - "C" and Delta - "B" values, where the ratio of the positive class is significantly low - less than 5% samples with positive class.

We can explore this more, but let's focus on the Epsilon attribute. It's the only one time-distributed variable and depicts the date of data collection. In this case, it's good to see the Class trend in time.

[47]:
```
rolling_mean_class = (
    greeks[["Epsilon", "Class"]]
    .assign(Epsilon=pd.to_datetime(greeks.Epsilon, errors="coerce"))
    .dropna()
    .sort_values(by="Epsilon")
    .rolling(window="365D", on="Epsilon")
    .mean()
)
```

```
fig = px.line(
    rolling_mean_class,
    x="Epsilon",
    y="Class",
    height=540,
    width=840,
    color_discrete_sequence=["#010D36"],
    symbol_sequence=["x"],
    line_shape="spline",
    markers=True,
    title="Class Trend - Rolling Mean over 365 Days",
)
fig.update_layout(
    font_color=FONT_COLOR,
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
)
fig.update_traces(marker=dict(size=6, color="#FF2079", opacity=0.7))
fig.show()
```

Observations

Well, most of the samples were collected from the end of 2018 up to 2020 autumn. We observe a relevant decreasing trend in the positive class target within samples collected from the end of 2018 to begin of 2019. This feature looks like a powerful predictive variable. However, it's available only for the training set, so including time in the learning process may be risky.

[48]:
```
greeks["Epsilon Availability"] = (greeks.Epsilon != "Unknown").map(
    {True: "Epsilon Available", False: "Epsilon Missing"}
)

fig = px.sunburst(
    greeks.assign(Class=greeks.Class.map({0: "Class 0", 1: "Class 1"})),
    title="Class vs Epsilon Availability",
    path=["Epsilon Availability", "Class"],
    color_discrete_sequence=["#010D36", "#FF2079"],
    height=640,
    width=640,
)
fig.update_traces(
    insidetextorientation="horizontal",
    texttemplate="%{label}<br>%{value} - %{percentParent}",
    marker_line_width=5,
    marker_line_color=BACKGROUND_COLOR,
)
fig.update_layout(
    font_color=FONT_COLOR,
```

```
    title_font_size=18,
    plot_bgcolor=BACKGROUND_COLOR,
    paper_bgcolor=BACKGROUND_COLOR,
)
fig.show()
```

Observations

The unknown time of sample collection is only associated with the negative class target, thus machine learning models can relate that, which is not exactly what we want. Additionally, accurate imputation is probably not possible here.

[49]:
```
df = greeks[["Epsilon", "Class"]]
df.Epsilon = (
    pd.to_datetime(df.Epsilon, errors="coerce")
    .apply(pd.Timestamp.toordinal)
    .replace(1, np.nan)
    .transform(lambda x: (x - x.min()) / (x.max() - x.min()))
)
df = df.dropna()

epsilon = df.Epsilon.to_numpy()[:, np.newaxis]
target = df.Class.to_numpy()

mutual_info = mutual_info_classif(epsilon, target, random_state=42)
f_stat, p_value = f_classif(epsilon, target)
```

[50]:
```
print(CLR + "Mutual Information: ", RED + f"{mutual_info[0]:.2f}")
print(
    CLR + "ANOVA Test - F-statistic: ",
    RED + f"{f_stat[0]:.2f}",
)
print(
    CLR + "ANOVA Test - p-value associated with the F-statistic: ",
    RED + f"{p_value[0]:.2e}",
)
```

```
Mutual Information:  0.18

ANOVA Test - F-statistic:  25.89

ANOVA Test - p-value associated with the F-statistic:

5.24e-07
```

Observations

Mutual information is a measure which estimates the relationship between two random variables, which are simultaneously sampled. Intuitively it can be understood how much one variable tells us about another one. Mutual information is equal to zero if and only if two variables are statistically independent. We got a value of around 0.18, which is much greater than for the DU feature (0.07),

which was at the top in the mutual information test for available features concerning the target class.

In the ANOVA test, the null hypothesis is that there is no relationship between the feature and the target variable. The p-value we got is extremely small, which indicates strong evidence against the null hypothesis. Typically if the p-value is smaller than 0.05, we should consider it as statistically significant.

To summarize, Epsilon has a statistically significant relationship with the Class.

#

Preprocessing Pipeline

Notes

In preprocessing, we use transformations we've found and binarization for semi-constant variables. Missing values in continuous features are filled with KNN imputation.

[51]:
```
semi_const_cols = semi_const_cols_thresholds.keys()

# We don't have square root transformations.
no_transform_cols = no_transform_cols.drop(semi_const_cols, errors="ignore")
log_transform_cols = log_transform_cols.drop(semi_const_cols, errors="ignore")
reciprocal_transform_cols = reciprocal_transform_cols.drop(semi_const_cols,␣
 ↪errors="ignore")
boxcox_transform_cols = boxcox_transform_cols.drop(semi_const_cols,␣
 ↪errors="ignore")
yeojohnson_transform_cols = yeojohnson_transform_cols.drop(semi_const_cols,␣
 ↪errors="ignore")

preliminary_preprocess = make_pipeline(
    make_column_transformer(
        (
            StandardScaler(),
            no_transform_cols.to_list(),
        ),
        (
            make_pipeline(
                FunctionTransformer(func=np.log,␣
 ↪feature_names_out="one-to-one"),
                StandardScaler(),
            ),
            log_transform_cols.to_list(),
        ),
        (
            make_pipeline(
                FunctionTransformer(func=np.reciprocal,␣
 ↪feature_names_out="one-to-one"),
                StandardScaler(),
```

```
            ),
            reciprocal_transform_cols.to_list(),
        ),
        (
            PowerTransformer(method="box-cox", standardize=True),
            boxcox_transform_cols.to_list(),
        ),
        (
            PowerTransformer(method="yeo-johnson", standardize=True),
            yeojohnson_transform_cols.to_list(),
        ),
        (
            make_pipeline(
                SimpleImputer(strategy="most_frequent"),
                OrdinalEncoder(handle_unknown="use_encoded_value",␣
 ↪unknown_value=-1),
            ),
            make_column_selector(dtype_include=object),  # type: ignore
        ),
        *[
            (
                make_pipeline(
                    SimpleImputer(strategy="median"),
                    Binarizer(threshold=thresh),
                ),
                [col],
            )
            for col, thresh in semi_const_cols_thresholds.items()
        ],
        remainder="drop",
        verbose_feature_names_out=False,
    ),
    KNNImputer(n_neighbors=10, weights="distance"),
).set_output(transform="pandas")
```

[52]:
```
X_preliminary = preliminary_preprocess.fit_transform(train.drop("Class",␣
 ↪axis=1))

assert np.all(np.isfinite(X_preliminary)) == True
assert np.any(np.isnan(X_preliminary)) == False

X_preliminary.head().style.set_table_styles(DF_STYLE).format(precision=3)
```

[52]:
```
<pandas.io.formats.style.Styler at 0x7b3330a89e10>
```

[53]:
```
print(
    CLR + "Training dataset shape before preprocessing:",
```

```
    RED + f"{train.drop('Class', axis=1).shape}",
)
print(
    CLR + "Training dataset shape after preprocessing: ",
    RED + f"{X_preliminary.shape}",
)
```

```
Training dataset shape before preprocessing: (617, 56)
Training dataset shape after preprocessing:  (617, 56)
```

Observations

Everything should work fine.