

Introduction

Business Problem

Dataset: Subscription data from a marketing campaign conducted by a banking institution.

Problem Statement: The business problem is a binary classification challenge. The goal of the classification is to predict whether a client contacted through the marketing campaign will subscribe to a term deposit.

Workflow

- Exploratory Data Analysis (EDA)
- Preprocessing (including dimensionality reduction and feature engineering)
- Identification of Most Important Features
- Model Development (including hyperparameter tuning and boosting)
- Model Evaluation
- End-to-End Process Deployment 😊

```
In [62]: import numpy as np
import re
import pandas as pd
import seaborn as sns
import plotly.io as pio
import plotly.express as px
import plotly.graph_objects as go
import matplotlib.pyplot as plt

from imblearn.over_sampling import SMOTE
from sklearn.metrics import r2_score, accuracy_score, precision_score, recall_score
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import LabelEncoder, StandardScaler, OneHotEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, KFold, cross_val_score
from sklearn.metrics import classification_report
from xgboost import XGBClassifier
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

file_path = '/file/path/to/your/file/'
file_name = '/your/file/name.csv'
target = 'your_target'
```

```
In [2]: # from google.colab import drive
# drive.mount('/content/drive')
```

Mounted at /content/drive

Input variables:

Bank client data:

1. **age**: (numeric)
2. **job**: type of job (categorical)
3. **marital status**: (categorical)
4. **education**: (categorical)
5. **default**: has credit in default? (categorical)
6. **housing**: has a housing loan? (categorical)
7. **loan**: has a personal loan? (categorical)

Related to the last contact of the current campaign:

8. **contact**: contact communication type (categorical)
9. **month**: last contact month of the year (categorical)
10. **day of week**: last contact day of the week (categorical)
11. **duration**: last contact duration in seconds (numeric)

***Other attributes:**

12. **campaign**: number of contacts performed during this campaign for this client (numeric; includes last contact)
13. **pdays**: number of days since the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
14. **previous**: number of contacts performed before this campaign for this client (numeric)
15. **poutcome**: outcome of the previous marketing campaign (categorical)

Social and economic context attributes:

16. **employment variation rate**: quarterly indicator (numeric)
17. **consumer price index**: monthly indicator (numeric)
18. **consumer confidence index**: monthly indicator (numeric)
19. **euribor 3 month rate**: daily indicator (numeric)
20. **number of employees**: quarterly indicator (numeric)

Output variable (target):

21. **y**: has the client subscribed to a term deposit? (binary)

Feature from initial observations

duration (feature 11): "this represents the length of the last contact in seconds (numeric)."

Why is duration important?

- **direct impact:** the length of the last call is a strong indicator of whether a client will subscribe to the term deposit. generally, longer conversations often reflect more client interest.
- **empirical insight:** a duration of zero usually signals a lack of client interest, leading to a higher likelihood of a "no" outcome.
- **benchmark usage:** while in real-world scenarios, the duration shouldn't be used in the model, it acts as a powerful benchmark predictor, highlighting its significance despite potential biases.

Other key features to watch:

although duration is the most influential feature, there are others that also play an important role:

- **previous campaign outcome (feature 15: poutcome):** the result of past campaigns can provide insights into a client's likelihood of subscribing.
- **number of contacts (feature 12: campaign):** repeated contact with the client may influence their decision-making.
- **employment variation rate (feature 16: emp.var.rate):** changes in employment rates can affect decisions about financial investments like term deposits.
- **euribor 3 month rate (feature 19: euribor3m):** fluctuating interest rates can make term deposits more or less attractive. while duration stands out as a key predictor, especially for benchmark purposes, if it's excluded, consider focusing on the combined influence of **poutcome**, **emp.var.rate**, and **campaign** for predictive analysis. **note:** duration has a significant influence on the target variable (e.g., if the duration is zero, the likely outcome is "no"). however, since this value isn't known before a call, duration should only be used for benchmark purposes and omitted from actual predictive modeling.

The provided code defines two functions: `load_and_clean_csv`, which loads a CSV file into a `DataFrame`, cleans the column names and values by removing special characters and converting them to lowercase, and `summarize_and_unique_values`, which prints summary information of the `DataFrame` and returns unique values for each column, excluding a specified target column. The code then loads and cleans a CSV file using the first function and summarizes it while extracting unique values from the `DataFrame` using the second function. This workflow

facilitates data preprocessing and provides insights into the dataset's structure and content.

```
In [63]: def load_and_clean_csv(file_path: str, file_name: str) -> pd.DataFrame:
        """
        Load a CSV file into a DataFrame and clean column names and values by removing special characters
        and converting them to lowercase.
        Parameters:
        - file_path (str): Path to the directory containing the file.
        - file_name (str): The name of the CSV file to load.
        Returns:
        - DataFrame: Cleaned DataFrame with special characters removed from column names and values.
        """
        # Load CSV file
        df = pd.read_csv(f"{file_path}{file_name}", sep=';', encoding='utf-8')
        # Define the regex pattern for special characters
        pattern = r'^[a-zA-Z0-9\s]+'
        # Clean and convert column names to lowercase
        df.columns = [re.sub(pattern, '', col).strip().lower() for col in df.columns]
        # Clean object type column values and convert to lowercase
        for col in df.select_dtypes(include='object').columns:
            df[col] = df[col].apply(lambda x: re.sub(pattern, '', str(x)).strip().lower())
        return df

def summarize_and_unique_values(df: pd.DataFrame, target: str) -> dict:
    """
    Print summary information of the DataFrame and return unique values per column, excluding the target column.
    Parameters:
    - df (DataFrame): The input DataFrame to summarize and extract unique values from.
    - target (str): The target column to exclude from unique value extraction.
    Returns:
    - dict: A dictionary containing unique values for each column excluding the target column.
    """
    # Print DataFrame summary information
    print(df.info())
    print(f"\nThe sum of null values per column:\n{df.isnull().sum()}")
    # Return unique values for each column, excluding the target column
    return {col: df[col].unique().tolist() for col in df.columns if col != target}

# Load and clean the CSV file
df = load_and_clean_csv(file_path, file_name)
# Summarize the DataFrame and get unique values (excluding the target column)
unique_values = summarize_and_unique_values(df, target)
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
#   Column                Non-Null Count  Dtype
---  -
0   age                    41188 non-null  int64
1   job                    41188 non-null  object
2   marital                41188 non-null  object
3   education              41188 non-null  object
4   default                41188 non-null  object
5   housing                41188 non-null  object
6   loan                   41188 non-null  object
7   contact                41188 non-null  object
8   month                  41188 non-null  object
9   dayofweek              41188 non-null  object
10  duration               41188 non-null  int64
11  campaign               41188 non-null  int64
12  pdays                 41188 non-null  int64
13  previous               41188 non-null  int64
14  poutcome               41188 non-null  object
15  empvarrate             41188 non-null  float64
16  conspriceidx           41188 non-null  float64
17  consconfidx            41188 non-null  float64
18  euribor3m              41188 non-null  float64
19  nremployed             41188 non-null  float64
20  y                      41188 non-null  object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
None

```

The sum of null values per column:

```

age          0
job          0
marital      0
education    0
default      0
housing      0
loan         0
contact      0
month        0
dayofweek    0
duration     0
campaign     0
pdays       0
previous     0
poutcome     0
empvarrate   0
conspriceidx 0
consconfidx  0
euribor3m    0
nremployed   0
y            0
dtype: int64

```

Printing unique values for an initial overview can help identify potential missing needs.

```
In [92]: # Print unique values for inspection
print(f"\nUnique values per column (excluding '{target}'):")
for column, values in unique_values.items():
    print(f"{column}: {values}")
```

```

Unique values per column (excluding 'y'):
age: [56, 57, 37, 40, 45, 59, 41, 24, 25, 29, 35, 54, 46, 50, 39, 30, 55, 4
9, 34, 52, 58, 32, 38, 44, 42, 60, 53, 47, 51, 48, 33, 31, 43, 36, 28, 27, 2
6, 22, 23, 20, 21, 61, 19, 18, 70, 66, 76, 67, 73, 88, 95, 77, 68, 75, 63, 8
0, 62, 65, 72, 82, 64, 71, 69, 78, 85, 79, 83, 81, 74, 17, 87, 91, 86, 98, 9
4, 84, 92, 89]
job: ['other', 'technical', 'office', 'busyness', 'unknown']
marital: ['married', 'single', 'divorced', 'unknown']
education: ['basic4y', 'highschool', 'basic6y', 'basic9y', 'professionalcour
se', 'unknown', 'universitydegree', 'illiterate']
default: ['no', 'unknown', 'yes']
housing: ['no', 'yes', 'unknown']
loan: ['no', 'yes', 'unknown']
contact: ['telephone', 'cellular']
month: ['may', 'jun', 'jul', 'aug', 'oct', 'nov', 'dec', 'mar', 'apr', 'se
p']
dayofweek: ['mon', 'tue', 'wed', 'thu', 'fri']
duration: [261, 149, 226, 151, 307, 198, 139, 217, 380, 50, 55, 222, 137, 29
3, 146, 174, 312, 440, 353, 195, 38, 262, 342, 181, 172, 99, 93, 233, 255, 3
62, 348, 386, 73, 230, 208, 336, 365, 1666, 577, 366, 314, 160, 212, 188, 2
2, 616, 178, 355, 225, 266, 253, 179, 269, 135, 161, 787, 145, 449, 812, 16
4, 357, 232, 91, 273, 158, 177, 200, 176, 211, 214, 1575, 349, 337, 272, 19
3, 165, 1042, 20, 246, 529, 192, 1467, 180, 48, 213, 545, 583, 221, 426, 28
7, 197, 257, 229, 400, 190, 21, 300, 123, 325, 514, 849, 194, 286, 247, 518,
364, 98, 439, 79, 175, 61, 78, 102, 579, 143, 677, 267, 345, 185, 207, 69, 1
00, 125, 461, 240, 70, 136, 528, 541, 338, 163, 87, 301, 46, 52, 204, 155, 7
1, 243, 186, 559, 2033, 85, 506, 114, 843, 427, 292, 128, 107, 303, 81, 270,
228, 673, 250, 130, 252, 138, 412, 19, 717, 313, 289, 683, 1077, 167, 356, 2
77, 218, 67, 291, 248, 256, 477, 611, 471, 381, 251, 408, 322, 216, 210, 28
8, 168, 132, 64, 209, 410, 580, 127, 189, 238, 124, 18, 730, 40, 142, 389, 7
02, 117, 370, 119, 361, 350, 150, 332, 58, 89, 152, 110, 463, 962, 10, 118,
92, 75, 935, 56, 5, 206, 446, 742, 120, 122, 215, 205, 83, 106, 108, 358, 45
3, 173, 241, 224, 148, 199, 196, 111, 231, 316, 669, 425, 121, 88, 402, 144,
220, 254, 503, 680, 421, 113, 347, 404, 396, 379, 306, 77, 54, 344, 202, 27
8, 184, 235, 290, 133, 318, 437, 501, 1201, 1030, 769, 442, 455, 424, 43, 15
4, 393, 203, 140, 326, 483, 259, 227, 576, 90, 505, 245, 623, 496, 276, 744,
271, 141, 264, 309, 1623, 101, 354, 451, 159, 170, 112, 53, 134, 678, 182, 1
62, 27, 699, 1677, 310, 47, 30, 472, 116, 448, 169, 157, 49, 374, 531, 153,
80, 568, 918, 82, 166, 369, 371, 263, 41, 13, 26, 792, 242, 268, 375, 383, 1
297, 502, 260, 105, 524, 352, 695, 76, 535, 390, 315, 36, 1906, 219, 147, 40
7, 65, 284, 285, 258, 635, 802, 57, 304, 392, 201, 329, 328, 191, 532, 416,
37, 530, 29, 311, 507, 333, 739, 339, 308, 467, 378, 1597, 346, 60, 716, 23
4, 296, 283, 109, 95, 31, 593, 631, 32, 1529, 800, 239, 42, 305, 343, 126, 2
49, 59, 51, 275, 479, 96, 720, 395, 629, 131, 298, 97, 104, 852, 294, 74, 99
2, 464, 732, 359, 274, 1521, 615, 327, 236, 492, 1138, 295, 591, 786, 388, 2
5, 401, 435, 423, 799, 45, 68, 444, 223, 566, 376, 511, 866, 1581, 279, 129,
432, 516, 617, 171, 614, 485, 406, 650, 590, 72, 474, 1101, 912, 1062, 688,
103, 607, 331, 398, 803, 481, 418, 24, 441, 1009, 550, 764, 1273, 1574, 62,
517, 299, 244, 548, 66, 984, 1689, 84, 489, 865, 281, 944, 319, 35, 17, 280,
156, 813, 94, 183, 604, 86, 11, 405, 462, 39, 187, 323, 521, 1119, 12, 1120,
33, 784, 665, 475, 63, 712, 1007, 237, 500, 789, 513, 468, 756, 14, 491, 44,
989, 1170, 807, 534, 28, 302, 2087, 767, 627, 403, 626, 23, 543, 1178, 422,
15, 335, 956, 459, 4, 985, 672, 8, 330, 399, 297, 886, 341, 515, 1187, 466,
826, 598, 584, 847, 659, 772, 929, 710, 498, 705, 480, 2462, 1132, 384, 825,
490, 115, 646, 653, 377, 544, 324, 391, 654, 1087, 557, 1692, 622, 2016, 105
4, 282, 409, 1713, 551, 663, 1080, 1461, 750, 488, 460, 878, 317, 834, 1534,

```

836, 1002, 592, 757, 523, 363, 1147, 486, 539, 820, 788, 832, 1111, 1495, 49
3, 457, 891, 1083, 1266, 470, 793, 413, 574, 596, 320, 484, 456, 334, 504, 9
07, 723, 1346, 520, 382, 1386, 428, 360, 3366, 1000, 618, 351, 2231, 373, 34
0, 1167, 609, 806, 766, 1015, 768, 473, 1001, 845, 853, 452, 916, 443, 431,
565, 753, 708, 265, 434, 805, 3, 420, 367, 394, 411, 34, 851, 1052, 647, 77
1, 1093, 1106, 945, 816, 1721, 1032, 735, 438, 942, 387, 476, 606, 824, 132
8, 686, 1125, 1321, 858, 546, 429, 869, 833, 829, 749, 1028, 977, 927, 762,
746, 1044, 668, 726, 634, 554, 436, 902, 594, 636, 738, 482, 567, 582, 1118,
837, 1423, 856, 747, 1013, 415, 552, 644, 558, 1088, 1074, 1036, 397, 599, 1
257, 1165, 651, 734, 417, 587, 920, 1244, 719, 597, 525, 815, 911, 465, 973,
561, 1224, 589, 964, 1156, 1231, 619, 1051, 419, 1867, 760, 1263, 770, 487,
697, 430, 809, 7, 850, 855, 875, 892, 512, 601, 844, 676, 656, 1252, 1143, 7
31, 754, 679, 1230, 894, 703, 433, 1340, 897, 718, 1161, 16, 2680, 698, 112
8, 509, 1135, 1408, 827, 588, 522, 1193, 1144, 1023, 469, 385, 1245, 1064, 1
110, 882, 943, 798, 610, 1203, 1022, 643, 571, 445, 1622, 967, 1218, 3078, 1
205, 1882, 1334, 775, 600, 447, 1777, 774, 1313, 1452, 547, 1376, 1045, 625,
999, 657, 1063, 1446, 6, 919, 777, 1392, 725, 801, 938, 692, 905, 508, 783,
603, 872, 641, 958, 628, 494, 759, 819, 648, 951, 578, 795, 542, 828, 1307,
748, 563, 450, 899, 857, 660, 1681, 572, 573, 811, 890, 681, 1162, 1697, 86
0, 575, 987, 671, 713, 923, 700, 526, 621, 1349, 1171, 736, 785, 1073, 533,
924, 691, 536, 556, 1003, 926, 773, 893, 478, 553, 1438, 569, 372, 368, 105
9, 1222, 1034, 581, 974, 745, 630, 863, 1234, 642, 729, 895, 519, 321, 796,
724, 741, 414, 454, 896, 763, 633, 560, 955, 674, 740, 776, 537, 755, 751, 1
590, 709, 570, 0, 953, 3094, 1043, 662, 1168, 861, 1479, 1210, 821, 497, 118
3, 675, 694, 664, 864, 1730, 667, 1277, 585, 620, 1196, 733, 791, 1207, 936,
932, 879, 1026, 689, 1047, 637, 685, 1611, 752, 1185, 900, 814, 859, 1109, 6
45, 2260, 711, 555, 867, 652, 682, 854, 1269, 868, 1097, 1500, 1236, 613, 12
12, 1980, 722, 510, 3631, 947, 1075, 527, 1068, 658, 562, 758, 966, 612, 133
0, 930, 1576, 605, 1173, 963, 941, 1025, 495, 2456, 1259, 1363, 1516, 1336,
1242, 1141, 638, 1449, 1254, 2203, 624, 870, 910, 1149, 701, 761, 1053, 100
5, 690, 1084, 586, 983, 817, 1018, 884, 1011, 939, 1072, 1276, 1114, 1994, 8
62, 1567, 968, 1041, 1288, 639, 2653, 1085, 1271, 1469, 1291, 540, 1055, 109
8, 901, 940, 782, 952, 1137, 458, 706, 1199, 950, 838, 904, 649, 781, 885, 2
025, 993, 871, 1268, 602, 640, 1618, 1243, 1323, 1395, 822, 1238, 1298, 108
9, 1021, 1248, 721, 2769, 881, 714, 986, 1848, 1345, 2621, 979, 1208, 835, 1
528, 1487, 1540, 632, 707, 1255, 2093, 704, 1195, 1066, 1060, 1082, 810, 131
8, 922, 1411, 2028, 1136, 1017, 549, 1012, 9, 2635, 1573, 1663, 1617, 1478,
1422, 818, 693, 804, 1094, 988, 670, 655, 790, 3183, 780, 1049, 1992, 957, 9
91, 538, 1434, 1272, 1103, 1356, 889, 1200, 933, 830, 564, 1046, 1767, 1027,
831, 1720, 1061, 1344, 687, 2122, 990, 727, 1121, 1139, 728, 1317, 1014, 129
0, 1439, 1426, 1019, 1065, 1102, 982, 880, 1341, 2029, 1499, 839, 1399, 364
3, 794, 684, 906, 1294, 965, 970, 1973, 1389, 666, 1164, 595, 1081, 1180, 16
49, 1310, 1397, 1153, 1130, 1669, 1071, 1056, 778, 808, 1615, 981, 1228, 77
9, 1424, 1142, 1412, 715, 1806, 1150, 873, 1432, 1339, 1473, 1275, 1008, 158
4, 1448, 1151, 1390, 1319, 1175, 1673, 903, 978, 1503, 1127, 909, 1360, 137
3, 913, 1425, 1105, 1039, 848, 1877, 1342, 1134, 908, 898, 1352, 1545, 1833,
1220, 1281, 1029, 1508, 661, 608, 1206, 1237, 1287, 1148, 1037, 1226, 1608,
1176, 1152, 874, 1960, 1331, 1776, 1327, 1090, 2078, 1124, 1309, 1300, 1359,
1204, 1756, 1441, 1491, 915, 1869, 823, 1850, 1820, 1602, 1492, 1946, 743, 2
015, 1031, 1368, 946, 1369, 1169, 1096, 499, 1076, 1569, 2516, 797, 1186, 10
58, 1140, 1078, 1211, 2692, 921, 1739, 846, 1488, 976, 1311, 1332, 1227, 119
7, 996, 1357, 2191, 1552, 1250, 1471, 1456, 1462, 1834, 3422, 876, 1934, 130
6, 994, 765, 1070, 917, 1099, 1504, 1282, 971, 1181, 1133, 696, 1092, 1613,
1735, 1476, 1842, 969, 737, 1038, 1217, 3322, 1184, 1579, 1871, 1126, 1364,
1559, 1293, 1241, 1740, 975, 1464, 4199, 1505, 1532, 1117, 1123, 1258, 1329,
1222, 1266, 961, 1374, 1223, 2089, 1033, 1129, 1642, 2372, 3253, 2429, 3284,

1239, 4918, 1303, 1606, 1091, 960, 1978, 1122, 1855, 2, 1437, 1788, 972, 155
4, 925, 998, 1057, 1548, 1283, 1502, 1265, 1662, 1468, 1337, 1435, 997, 119
2, 1816, 1040, 1256, 931, 1490, 1154, 1035, 888, 1166, 1145, 2420, 1598, 245
3, 1221, 1182, 1480, 980, 1571, 1555, 1067, 1447, 3076, 1530, 2870, 2316, 13
53, 2299, 1, 2129, 1190, 1665, 1594, 1372, 1214, 1174, 1463, 1365, 1285, 141
0, 1095, 840, 1624, 949, 2926, 2053, 2139, 1202, 1112, 1550, 1010, 1817, 133
3, 1348, 1260, 1958, 1370, 841, 1240, 1100, 954, 1079, 1131, 1489, 1347, 195
7, 1191, 1391, 877, 1262, 1954, 1024, 1514, 934, 1232, 1388, 1531, 1925, 171
0, 1108, 1512, 1966, 937, 1970, 1302, 1975, 1805, 1279, 1020, 1723, 1543, 95
9, 1286, 1380, 1326, 2301, 1880, 3509, 1460, 1048, 2219, 1361, 1603, 883, 12
25, 1416, 1398, 928, 2055, 1962, 1104, 1551, 1580, 1745, 2187, 1707, 1233, 2
184, 1628, 1804, 2062, 1472, 2486, 1267, 1563, 1407, 2035, 3785, 1440, 1394,
1405, 1640, 1616, 1246, 1556, 1868]

campaign: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 19, 18, 23, 14, 22, 2
5, 16, 17, 15, 20, 56, 39, 35, 42, 28, 26, 27, 32, 21, 24, 29, 31, 30, 41, 3
7, 40, 33, 34, 43]

pdays: [-1, 6, 4, 3, 5, 1, 0, 10, 7, 8, 9, 11, 2, 12, 13, 14, 15, 16, 21, 1
7, 18, 22, 25, 26, 19, 27, 20]

previous: [0, 1, 2, 3, 4, 5, 6, 7]

poutcome: ['nonexistent', 'failure', 'success']

empvarrate: [1.1, 1.4, -0.1, -0.2, -1.8, -2.9, -3.4, -3.0, -1.7, -1.1]

conspriceidx: [93.994, 94.465, 93.918, 93.444, 93.798, 93.2, 92.756, 92.843,
93.075, 92.893, 92.963, 92.469, 92.201, 92.379, 92.431, 92.649, 92.713, 93.3
69, 93.749, 93.876, 94.055, 94.215, 94.027, 94.199, 94.601, 94.767]

consconfidx: [-36.4, -41.8, -42.7, -36.1, -40.4, -42.0, -45.9, -50.0, -47.1,
-46.2, -40.8, -33.6, -31.4, -29.8, -26.9, -30.1, -33.0, -34.8, -34.6, -40.0,
-39.8, -40.3, -38.3, -37.5, -49.5, -50.8]

euribor3m: [4.857, 4.856, 4.855, 4.859, 4.86, 4.858, 4.864, 4.865, 4.866, 4.
967, 4.961, 4.959, 4.958, 4.96, 4.962, 4.955, 4.947, 4.956, 4.966, 4.963, 4.
957, 4.968, 4.97, 4.965, 4.964, 5.045, 5.0, 4.936, 4.921, 4.918, 4.912, 4.82
7, 4.794, 4.76, 4.733, 4.7, 4.663, 4.592, 4.474, 4.406, 4.343, 4.286, 4.245,
4.223, 4.191, 4.153, 4.12, 4.076, 4.021, 3.901, 3.879, 3.853, 3.816, 3.743,
3.669, 3.563, 3.488, 3.428, 3.329, 3.282, 3.053, 1.811, 1.799, 1.778, 1.757,
1.726, 1.703, 1.687, 1.663, 1.65, 1.64, 1.629, 1.614, 1.602, 1.584, 1.574,
1.56, 1.556, 1.548, 1.538, 1.531, 1.52, 1.51, 1.498, 1.483, 1.479, 1.466, 1.
453, 1.445, 1.435, 1.423, 1.415, 1.41, 1.405, 1.406, 1.4, 1.392, 1.384, 1.37
2, 1.365, 1.354, 1.344, 1.334, 1.327, 1.313, 1.299, 1.291, 1.281, 1.266, 1.2
5, 1.244, 1.259, 1.264, 1.27, 1.262, 1.26, 1.268, 1.286, 1.252, 1.235, 1.22
4, 1.215, 1.206, 1.099, 1.085, 1.072, 1.059, 1.048, 1.044, 1.029, 1.018, 1.0
07, 0.996, 0.979, 0.969, 0.944, 0.937, 0.933, 0.927, 0.921, 0.914, 0.908, 0.
903, 0.899, 0.884, 0.883, 0.881, 0.879, 0.873, 0.869, 0.861, 0.859, 0.854,
0.851, 0.849, 0.843, 0.838, 0.834, 0.829, 0.825, 0.821, 0.819, 0.813, 0.809,
0.803, 0.797, 0.788, 0.781, 0.778, 0.773, 0.771, 0.77, 0.768, 0.766, 0.762,
0.755, 0.749, 0.743, 0.741, 0.739, 0.75, 0.753, 0.754, 0.752, 0.744, 0.74,
0.742, 0.737, 0.735, 0.733, 0.73, 0.731, 0.728, 0.724, 0.722, 0.72, 0.719,
0.716, 0.715, 0.714, 0.718, 0.721, 0.717, 0.712, 0.71, 0.709, 0.708, 0.706,
0.707, 0.7, 0.655, 0.654, 0.653, 0.652, 0.651, 0.65, 0.649, 0.646, 0.644, 0.
643, 0.639, 0.637, 0.635, 0.636, 0.634, 0.638, 0.64, 0.642, 0.645, 0.659, 0.
663, 0.668, 0.672, 0.677, 0.682, 0.683, 0.684, 0.685, 0.688, 0.69, 0.692, 0.
695, 0.697, 0.699, 0.701, 0.702, 0.704, 0.711, 0.713, 0.723, 0.727, 0.729,
0.732, 0.748, 0.761, 0.767, 0.782, 0.79, 0.793, 0.802, 0.81, 0.822, 0.827,
0.835, 0.84, 0.846, 0.87, 0.876, 0.885, 0.889, 0.893, 0.896, 0.898, 0.9, 0.9
04, 0.905, 0.895, 0.894, 0.891, 0.89, 0.888, 0.886, 0.882, 0.88, 0.878, 0.87
7, 0.942, 0.953, 0.956, 0.959, 0.965, 0.972, 0.977, 0.982, 0.985, 0.987, 0.9
93, 1.0, 1.008, 1.016, 1.025, 1.032, 1.037, 1.043, 1.045, 1.047, 1.05, 1.04
0, 1.046, 1.041, 1.04, 1.039, 1.035, 1.03, 1.031, 1.028]

```
nemployed: [5191.0, 5228.1, 5195.8, 5176.3, 5099.1, 5076.2, 5017.5, 5023.5,
5008.7, 4991.6, 4963.6]
age_category: ['prime', 'young', 'veteran']
education_category: ['basic', 'higher', 'unknown', 'other']
duration_category: ['long', 'medium', 'short', 'very_long', 'very Short']
previous_contact: [0, 1]
campaign_category: ['low_engagement', 'moderate_engagement', 'high_engagemen
t']
```

Observation: After an initial overview of the unique values in the data, several potential issues are evident from a modeling perspective, where preprocessing and feature engineering are recommended.

- **job:** there are too many categories in this feature. recategorization will reduce the number of unique values, positively impacting computation.
- **age:** categorizing age into groups such as 'young', 'prime', and 'old' could improve model performance by providing clearer distinctions in age ranges.
- **pdays:** the value 999 is a significant outlier and should be processed (it will likely be replaced with -1).
- **campaign:** similar to age, categorizing the campaign feature could be beneficial.
- **duration:** this feature also requires categorization.
- **education:** similar to the job feature, categories such as 'basic4y', 'highschool', 'basic6y', and 'basic9y' can be grouped into a 'basic' category.

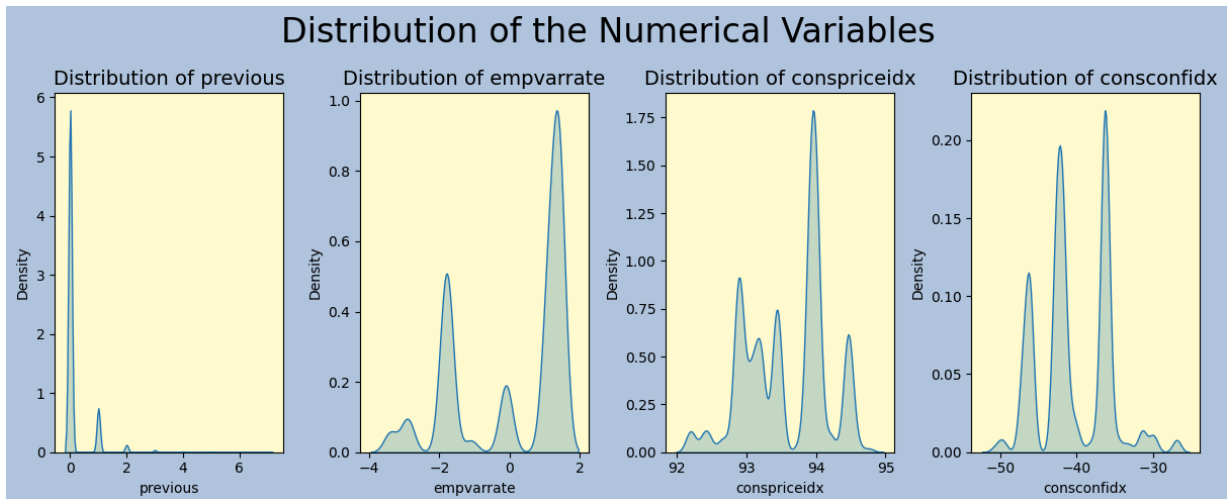
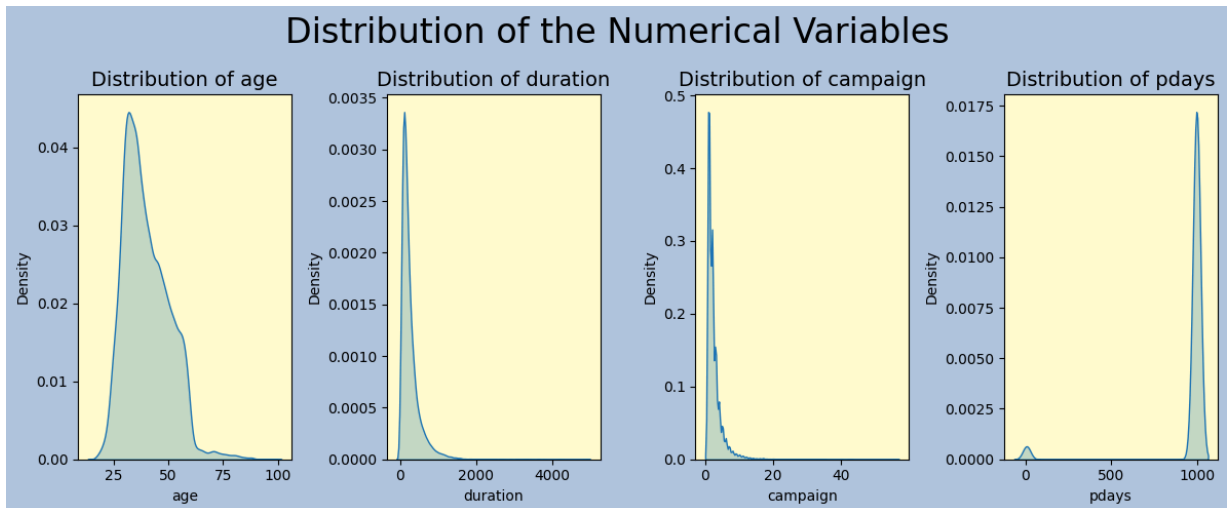
For categorical values, there are numerous distinct entries that could benefit from advanced categorization, such as 'basic4y', 'highschool', 'basic6y', and 'basic9y', which can be grouped under a single category labeled 'basic' education. This approach simplifies the dataset by reducing complexity and enhancing interpretability. Implementing such categorization can improve the effectiveness of subsequent analyses and modeling efforts. But first, lets performe global analysis

```
In [4]: def plot_distributions(df):
        """
        Plots distributions for numerical variables in the dataset.
        Parameters:
        data : pandas DataFrame
            The input data containing the features.
        """
        # Get the numerical-type columns
        numerical_features = df.select_dtypes(include=['number']).columns.tolist()
        # Calculate the number of columns and how many rows of plots will be needed
        n_cols = len(numerical_features)
        n_rows = (n_cols + 3) // 4 # 4 plots per row
        for i in range(n_rows):
            fig, axes = plt.subplots(ncols=4, nrows=1, figsize=(12, 5))
            for j in range(4):
                index = i * 4 + j
                if index < n_cols:
```

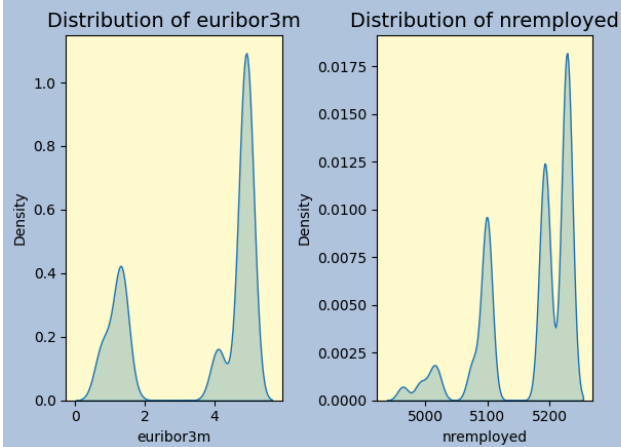
```

ax = axes[j] # Get the corresponding axis
# Using sns.kdeplot for smooth distribution
sns.kdeplot(df[numerical_features[index]], ax=ax, fill=True)
ax.set_title('Distribution of ' + str(numerical_features[index]))
ax.set_facecolor('lemonchiffon')
else:
    axes[j].axis('off') # Hide any unused axes
fig.suptitle("Distribution of the Numerical Variables", fontsize=24)
plt.tight_layout()
fig.set_facecolor('lightsteelblue')
plt.show()
plot_distributions(df)

```



Distribution of the Numerical Variables



Observation:

- **age**: the majority of clients are aged 30–40, with fewer clients in older age groups. the distribution is right-skewed, indicating a smaller proportion of older clients.
- **duration**: most calls were short, with very few long calls. the data is heavily skewed toward shorter durations.
- **campaign**: clients were typically contacted once or twice during the campaign. very few clients received multiple contacts.
- **pdays**: the value 999 dominates the distribution, showing that most clients had no previous contact. few other values are present, indicating sparse prior contact information.
- **previous**: most clients had no or very few previous contacts. the distribution is heavily skewed toward zero previous interactions.
- **employment variation rate (empvarrate)**: multiple peaks in the employment variation rate indicate fluctuations, reflecting different economic conditions.
- **consumer price index (conspiceidx)**: there are clear peaks in the consumer price index, showing distinct economic periods rather than a continuous trend.
- **consumer confidence index (consconfidx)**: the distribution shows multiple peaks, suggesting specific economic conditions and fluctuations in consumer confidence.
- **euribor 3-month rate**: most data points cluster around the 4% mark, with fewer observations at lower rates.
- **number of employees (nremployed)**: employment numbers are mostly concentrated around 5100–5200, showing stability in the workforce.

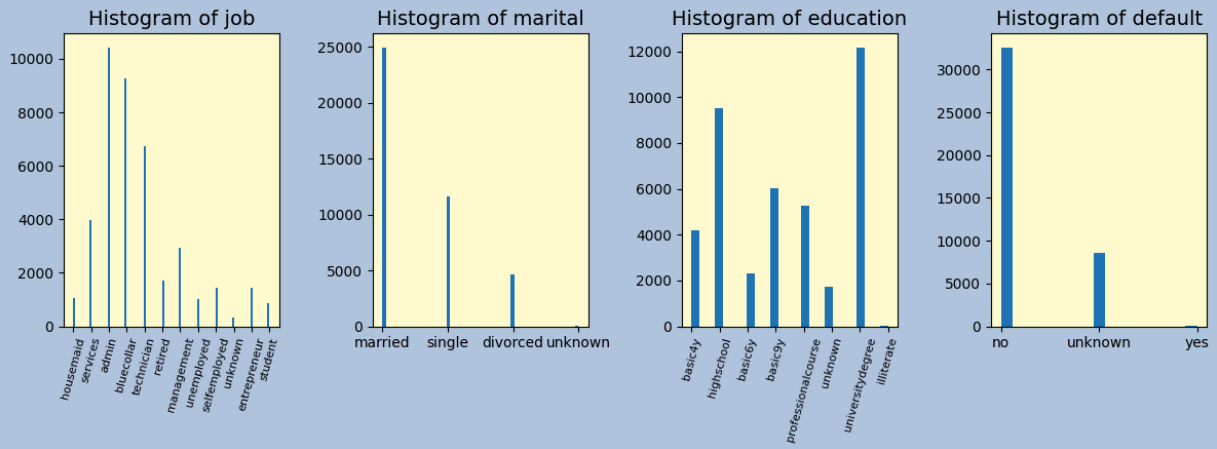
Overall conclusions:

- **age**: concentration of clients in the 30–40 age range, with fewer older clients.
- **duration**: calls are predominantly short in length.
- **campaign**: most clients were contacted once or twice, indicating low contact frequency.
- **previous contacts**: very few clients had previous interactions with the campaign.
- **economic indicators**: fluctuations are evident in employment rates, consumer price index, and consumer confidence index.
- **stable features**: euribor rates and employment numbers show stability across the dataset.

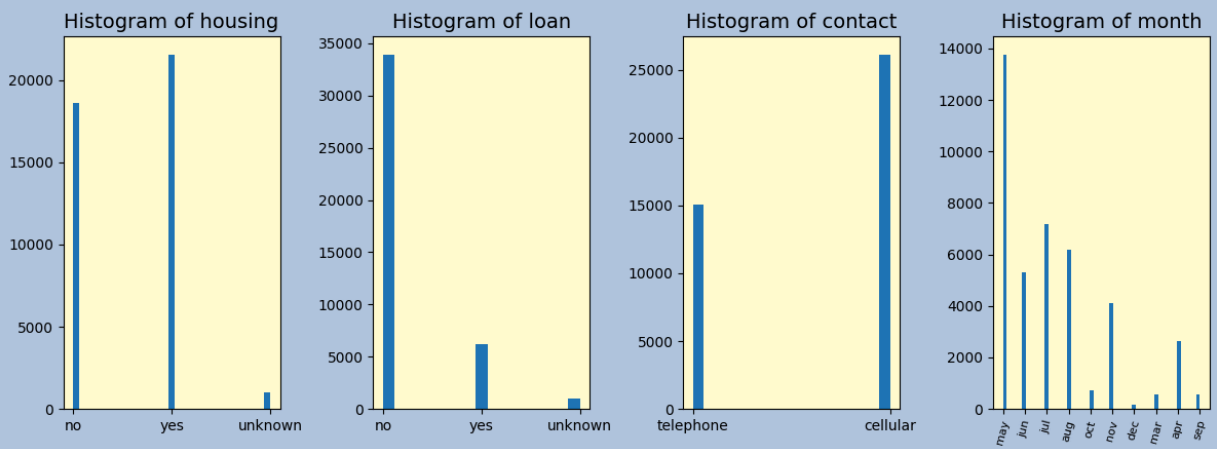
This overview suggests that the data will need preprocessing steps like handling skewness, outliers, and potentially converting some numerical variables into categorical or grouped values for better model performance.

```
In [ ]: def plot_histograms(df):
        """
        Plots histograms for categorical variables in the dataset.
        Parameters:
        data : pandas DataFrame
            The input data containing the features.
        """
        # Get the object-type columns
        object_features = df.select_dtypes(include=['object']).columns.tolist()
        # Calculate the number of columns and how many rows of plots will be needed
        n_cols = len(object_features)
        n_rows = (n_cols + 3) // 4 # 4 plots per row
        for i in range(n_rows):
            fig, axes = plt.subplots(ncols=4, nrows=1, figsize=(12, 5))
            for j in range(4):
                index = i * 4 + j
                if index < n_cols:
                    ax = axes[j] # Get the corresponding axis
                    ax.hist(df[object_features[index]], bins='auto') # Automati
                    ax.set_title('Histogram of ' + str(object_features[index]),
                                ax.set_facecolor('lemonchiffon')
                    # Adjust x-tick labels if there are more than 5 unique values
                    if df[object_features[index]].nunique() > 5:
                        ticks = ax.get_xticks() # Get the current ticks
                        ax.set_xticks(ticks) # Set ticks explicitly
                        ax.set_xticklabels(ax.get_xticklabels(), rotation=75, fo
                    else:
                        axes[j].axis('off') # Hide any unused axes
            fig.suptitle("Histograms of the Categorical Variables", fontsize=24)
            plt.tight_layout()
            fig.set_facecolor('lightsteelblue')
            plt.show()
        plot_histograms(df)
```

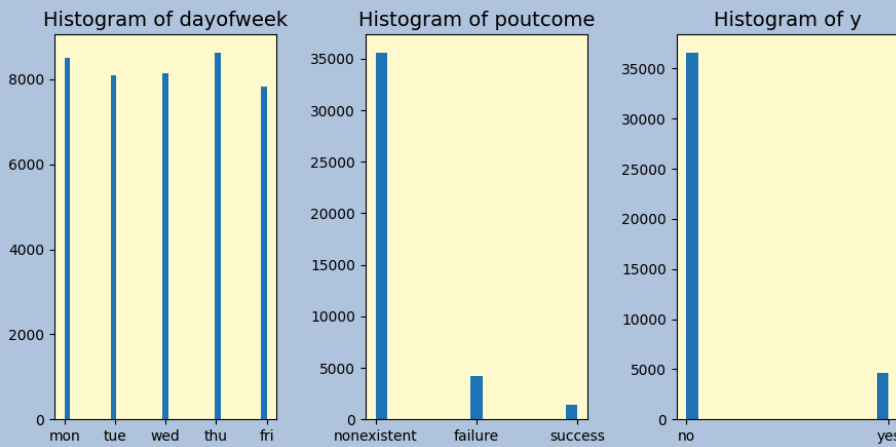
Histograms of the Categorical Variables



Histograms of the Categorical Variables



Histograms of the Categorical Variables



- **Job:** Majority in "bluecollar" and "management" roles, followed by "technicians" and "admin". Few "students", "unemployed" or "entrepreneurs".
- **Marital Status:** Mostly "married" then "single". Few "divorced" and almost no "unknown".
- **Education:** Dominated by "secondary" and "tertiary" education. Fewer with "primary" and "unknown" levels.

- **Default:** Most clients have no credit default. "unknown" category is larger than "yes".
- **Housing Loan:** Majority either have or don't have a housing loan; very few "unknown".
- **Personal Loan:** Most do not have a personal loan, with minimal "unknown".
- **Contact Type:** Predominantly via "cellular," fewer via "telephone".
- **Month:** Most contacts in May, followed by August and July. Fewer in March, September, and December.
- **Day of the Week:** Even distribution across weekdays.
- **Poutcome:** Most are "nonexistent" from previous campaigns, with "failure" being second and "success" rare.
- **Target Variable (y):** Most clients did not subscribe to a term deposit, with few who did.

Overall Conclusions:

- **Job:** Dominated by blue-collar and managerial roles.
- **Marital/Education:** Mostly married, with secondary/tertiary education.
- **Loans:** Few personal loans or credit defaults.
- **Contact:** Mostly via mobile phones, often in May, evenly spread across weekdays.
- **Campaign:** Most previous campaigns were unsuccessful, and few clients subscribed to the term deposit.

The DataPreprocessor class is created to help transform and categorize various features in a DataFrame, making it easier to analyze the data. It offers methods to categorize aspects like age, job, duration, education, and campaign engagement, along with processing the 'pdays' column and converting month and day values into numeric formats. The categories are hardcoded, so it is possible that modification will be needed

```
In [64]: # Separate coding for y cuz after simple coding yes:0 no:1 but must vise-ver
class DataPreprocessor:
    def __init__(self, df):
        """Initialize with the DataFrame to be processed."""
        self.df = df
    def categorize_age(self, age):
        """Categorize age into groups."""
        if age < 30:
            return 'young'
        elif 30 <= age < 65:
            return 'prime'
        else:
            return 'veteran'
    def categorize_job(self, job):
```

```

        """Categorize job into groups."""
        job_categories = {
            'admin': 'office', 'management': 'office',
            'technician': 'technical', 'services': 'technical', 'bluecollar':
            'entrepreneur': 'busyness', 'selfemployed': 'busyness',
            'unknown': 'unknown'
        }
        return job_categories.get(job, 'other')
def categorize_duration(self, duration):
    """Categorize duration into predefined categories."""
    if duration < 31:
        return 'very Short'
    elif 31 <= duration < 61:
        return 'short'
    elif 61 <= duration < 181:
        return 'medium'
    elif 181 <= duration < 601:
        return 'long'
    else:
        return 'very_long'
def process_pdays(self, column_name):
    """Process 'pdays' column and create a previous_contact binary column
    self.df[column_name] = self.df[column_name].replace(999, -1)
    self.df['previous_contact'] = (self.df[column_name] != -1).astype(int)
def categorize_education(self, education):
    if education in ['basic9y', 'basic6y', 'basic4y', 'highschool']:
        return 'basic'
    elif education in ['universitydegree', 'professionalcourse']:
        return 'higher'
    elif education in ['unknown']:
        return 'unknown'
    else:
        return 'other'
def categorize_campaign(self, campaign):
    """Categorize campaign engagement levels."""
    if campaign < 5:
        return 'low_engagement'
    elif 5 <= campaign < 15:
        return 'moderate_engagement'
    else:
        return 'high_engagement'
def map_columns(self):
    """Map month and day columns to numeric values."""
    month_mapping = {month: index + 1 for index, month in enumerate(['jan', 'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct', 'nov', 'dec'])}
    day_mapping = {day: index for index, day in enumerate(['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'])}
def preprocess(self):
    """Run all preprocessing steps."""
    self.df['age_category'] = self.df['age'].apply(self.categorize_age)
    self.df['job'] = self.df['job'].apply(self.categorize_job)
    self.df['education_category'] = self.df['education'].apply(self.categorize_education)
    self.df['duration_category'] = self.df['duration'].apply(self.categorize_duration)
    self.process_pdays('pdays')
    self.df['campaign_category'] = self.df['campaign'].apply(self.categorize_campaign)
    self.map_columns()

```

```

    return self.df # Return the modified DataFrame

```



```
preprocessor = DataPreprocessor(df)
df_processed = preprocessor.preprocess()

# Additional validation and unique values check
df_processed.head()
unique_values = summarize_and_unique_values(df_processed, target)
# Print unique values for inspection
print(f"\nUnique values per column (excluding '{target}'):")
for column, values in unique_values.items():
    print(f"{column}: {values}")
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 26 columns):
#   Column                Non-Null Count  Dtype
---  -
0   age                    41188 non-null  int64
1   job                    41188 non-null  object
2   marital                41188 non-null  object
3   education              41188 non-null  object
4   default                41188 non-null  object
5   housing                41188 non-null  object
6   loan                   41188 non-null  object
7   contact                41188 non-null  object
8   month                  41188 non-null  object
9   dayofweek              41188 non-null  object
10  duration                41188 non-null  int64
11  campaign                41188 non-null  int64
12  pdays                  41188 non-null  int64
13  previous                41188 non-null  int64
14  poutcome                41188 non-null  object
15  empvarrate              41188 non-null  float64
16  conspriceidx            41188 non-null  float64
17  consconfidx            41188 non-null  float64
18  euribor3m              41188 non-null  float64
19  nremployed              41188 non-null  float64
20  y                       41188 non-null  object
21  age_category            41188 non-null  object
22  education_category      41188 non-null  object
23  duration_category       41188 non-null  object
24  previous_contact        41188 non-null  int32
25  campaign_category       41188 non-null  object
dtypes: float64(5), int32(1), int64(5), object(15)
memory usage: 8.0+ MB
None

```

The sum of null values per column:

```

age                    0
job                    0
marital                0
education              0
default                0
housing                0
loan                   0
contact                0
month                  0
dayofweek              0
duration                0
campaign                0
pdays                  0
previous                0
poutcome                0
empvarrate              0
conspriceidx            0
consconfidx            0
euribor3m              0
nremployed              0
y                       0
age_category            0
education_category      0
duration_category       0
previous_contact        0
campaign_category       0

```

```

y                0
age_category     0
education_category 0
duration_category 0
previous_contact  0
campaign_category 0
dtype: int64

```

Unique values per column (excluding 'y'):

```

age: [56, 57, 37, 40, 45, 59, 41, 24, 25, 29, 35, 54, 46, 50, 39, 30, 55, 4
9, 34, 52, 58, 32, 38, 44, 42, 60, 53, 47, 51, 48, 33, 31, 43, 36, 28, 27, 2
6, 22, 23, 20, 21, 61, 19, 18, 70, 66, 76, 67, 73, 88, 95, 77, 68, 75, 63, 8
0, 62, 65, 72, 82, 64, 71, 69, 78, 85, 79, 83, 81, 74, 17, 87, 91, 86, 98, 9
4, 84, 92, 89]
job: ['other', 'technical', 'office', 'busyness', 'unknown']
marital: ['married', 'single', 'divorced', 'unknown']
education: ['basic4y', 'highschool', 'basic6y', 'basic9y', 'professionalcour
se', 'unknown', 'universitydegree', 'illiterate']
default: ['no', 'unknown', 'yes']
housing: ['no', 'yes', 'unknown']
loan: ['no', 'yes', 'unknown']
contact: ['telephone', 'cellular']
month: ['may', 'jun', 'jul', 'aug', 'oct', 'nov', 'dec', 'mar', 'apr', 'se
p']
dayofweek: ['mon', 'tue', 'wed', 'thu', 'fri']
duration: [261, 149, 226, 151, 307, 198, 139, 217, 380, 50, 55, 222, 137, 29
3, 146, 174, 312, 440, 353, 195, 38, 262, 342, 181, 172, 99, 93, 233, 255, 3
62, 348, 386, 73, 230, 208, 336, 365, 1666, 577, 366, 314, 160, 212, 188, 2
2, 616, 178, 355, 225, 266, 253, 179, 269, 135, 161, 787, 145, 449, 812, 16
4, 357, 232, 91, 273, 158, 177, 200, 176, 211, 214, 1575, 349, 337, 272, 19
3, 165, 1042, 20, 246, 529, 192, 1467, 180, 48, 213, 545, 583, 221, 426, 28
7, 197, 257, 229, 400, 190, 21, 300, 123, 325, 514, 849, 194, 286, 247, 518,
364, 98, 439, 79, 175, 61, 78, 102, 579, 143, 677, 267, 345, 185, 207, 69, 1
00, 125, 461, 240, 70, 136, 528, 541, 338, 163, 87, 301, 46, 52, 204, 155, 7
1, 243, 186, 559, 2033, 85, 506, 114, 843, 427, 292, 128, 107, 303, 81, 270,
228, 673, 250, 130, 252, 138, 412, 19, 717, 313, 289, 683, 1077, 167, 356, 2
77, 218, 67, 291, 248, 256, 477, 611, 471, 381, 251, 408, 322, 216, 210, 28
8, 168, 132, 64, 209, 410, 580, 127, 189, 238, 124, 18, 730, 40, 142, 389, 7
02, 117, 370, 119, 361, 350, 150, 332, 58, 89, 152, 110, 463, 962, 10, 118,
92, 75, 935, 56, 5, 206, 446, 742, 120, 122, 215, 205, 83, 106, 108, 358, 45
3, 173, 241, 224, 148, 199, 196, 111, 231, 316, 669, 425, 121, 88, 402, 144,
220, 254, 503, 680, 421, 113, 347, 404, 396, 379, 306, 77, 54, 344, 202, 27
8, 184, 235, 290, 133, 318, 437, 501, 1201, 1030, 769, 442, 455, 424, 43, 15
4, 393, 203, 140, 326, 483, 259, 227, 576, 90, 505, 245, 623, 496, 276, 744,
271, 141, 264, 309, 1623, 101, 354, 451, 159, 170, 112, 53, 134, 678, 182, 1
62, 27, 699, 1677, 310, 47, 30, 472, 116, 448, 169, 157, 49, 374, 531, 153,
80, 568, 918, 82, 166, 369, 371, 263, 41, 13, 26, 792, 242, 268, 375, 383, 1
297, 502, 260, 105, 524, 352, 695, 76, 535, 390, 315, 36, 1906, 219, 147, 40
7, 65, 284, 285, 258, 635, 802, 57, 304, 392, 201, 329, 328, 191, 532, 416,
37, 530, 29, 311, 507, 333, 739, 339, 308, 467, 378, 1597, 346, 60, 716, 23
4, 296, 283, 109, 95, 31, 593, 631, 32, 1529, 800, 239, 42, 305, 343, 126, 2
49, 59, 51, 275, 479, 96, 720, 395, 629, 131, 298, 97, 104, 852, 294, 74, 99
2, 464, 732, 359, 274, 1521, 615, 327, 236, 492, 1138, 295, 591, 786, 388, 2
5, 401, 435, 423, 799, 45, 68, 444, 223, 566, 376, 511, 866, 1581, 279, 129,
432, 516, 617, 171, 614, 485, 406, 650, 590, 72, 474, 1101, 912, 1062, 688,
102, 607, 331, 398, 803, 481, 418, 24, 441, 1009, 550, 764, 1273, 1574, 62,

```

517, 299, 244, 548, 66, 984, 1689, 84, 489, 865, 281, 944, 319, 35, 17, 280,
156, 813, 94, 183, 604, 86, 11, 405, 462, 39, 187, 323, 521, 1119, 12, 1120,
33, 784, 665, 475, 63, 712, 1007, 237, 500, 789, 513, 468, 756, 14, 491, 44,
989, 1170, 807, 534, 28, 302, 2087, 767, 627, 403, 626, 23, 543, 1178, 422,
15, 335, 956, 459, 4, 985, 672, 8, 330, 399, 297, 886, 341, 515, 1187, 466,
826, 598, 584, 847, 659, 772, 929, 710, 498, 705, 480, 2462, 1132, 384, 825,
490, 115, 646, 653, 377, 544, 324, 391, 654, 1087, 557, 1692, 622, 2016, 105
4, 282, 409, 1713, 551, 663, 1080, 1461, 750, 488, 460, 878, 317, 834, 1534,
836, 1002, 592, 757, 523, 363, 1147, 486, 539, 820, 788, 832, 1111, 1495, 49
3, 457, 891, 1083, 1266, 470, 793, 413, 574, 596, 320, 484, 456, 334, 504, 9
07, 723, 1346, 520, 382, 1386, 428, 360, 3366, 1000, 618, 351, 2231, 373, 34
0, 1167, 609, 806, 766, 1015, 768, 473, 1001, 845, 853, 452, 916, 443, 431,
565, 753, 708, 265, 434, 805, 3, 420, 367, 394, 411, 34, 851, 1052, 647, 77
1, 1093, 1106, 945, 816, 1721, 1032, 735, 438, 942, 387, 476, 606, 824, 132
8, 686, 1125, 1321, 858, 546, 429, 869, 833, 829, 749, 1028, 977, 927, 762,
746, 1044, 668, 726, 634, 554, 436, 902, 594, 636, 738, 482, 567, 582, 1118,
837, 1423, 856, 747, 1013, 415, 552, 644, 558, 1088, 1074, 1036, 397, 599, 1
257, 1165, 651, 734, 417, 587, 920, 1244, 719, 597, 525, 815, 911, 465, 973,
561, 1224, 589, 964, 1156, 1231, 619, 1051, 419, 1867, 760, 1263, 770, 487,
697, 430, 809, 7, 850, 855, 875, 892, 512, 601, 844, 676, 656, 1252, 1143, 7
31, 754, 679, 1230, 894, 703, 433, 1340, 897, 718, 1161, 16, 2680, 698, 112
8, 509, 1135, 1408, 827, 588, 522, 1193, 1144, 1023, 469, 385, 1245, 1064, 1
110, 882, 943, 798, 610, 1203, 1022, 643, 571, 445, 1622, 967, 1218, 3078, 1
205, 1882, 1334, 775, 600, 447, 1777, 774, 1313, 1452, 547, 1376, 1045, 625,
999, 657, 1063, 1446, 6, 919, 777, 1392, 725, 801, 938, 692, 905, 508, 783,
603, 872, 641, 958, 628, 494, 759, 819, 648, 951, 578, 795, 542, 828, 1307,
748, 563, 450, 899, 857, 660, 1681, 572, 573, 811, 890, 681, 1162, 1697, 86
0, 575, 987, 671, 713, 923, 700, 526, 621, 1349, 1171, 736, 785, 1073, 533,
924, 691, 536, 556, 1003, 926, 773, 893, 478, 553, 1438, 569, 372, 368, 105
9, 1222, 1034, 581, 974, 745, 630, 863, 1234, 642, 729, 895, 519, 321, 796,
724, 741, 414, 454, 896, 763, 633, 560, 955, 674, 740, 776, 537, 755, 751, 1
590, 709, 570, 0, 953, 3094, 1043, 662, 1168, 861, 1479, 1210, 821, 497, 118
3, 675, 694, 664, 864, 1730, 667, 1277, 585, 620, 1196, 733, 791, 1207, 936,
932, 879, 1026, 689, 1047, 637, 685, 1611, 752, 1185, 900, 814, 859, 1109, 6
45, 2260, 711, 555, 867, 652, 682, 854, 1269, 868, 1097, 1500, 1236, 613, 12
12, 1980, 722, 510, 3631, 947, 1075, 527, 1068, 658, 562, 758, 966, 612, 133
0, 930, 1576, 605, 1173, 963, 941, 1025, 495, 2456, 1259, 1363, 1516, 1336,
1242, 1141, 638, 1449, 1254, 2203, 624, 870, 910, 1149, 701, 761, 1053, 100
5, 690, 1084, 586, 983, 817, 1018, 884, 1011, 939, 1072, 1276, 1114, 1994, 8
62, 1567, 968, 1041, 1288, 639, 2653, 1085, 1271, 1469, 1291, 540, 1055, 109
8, 901, 940, 782, 952, 1137, 458, 706, 1199, 950, 838, 904, 649, 781, 885, 2
025, 993, 871, 1268, 602, 640, 1618, 1243, 1323, 1395, 822, 1238, 1298, 108
9, 1021, 1248, 721, 2769, 881, 714, 986, 1848, 1345, 2621, 979, 1208, 835, 1
528, 1487, 1540, 632, 707, 1255, 2093, 704, 1195, 1066, 1060, 1082, 810, 131
8, 922, 1411, 2028, 1136, 1017, 549, 1012, 9, 2635, 1573, 1663, 1617, 1478,
1422, 818, 693, 804, 1094, 988, 670, 655, 790, 3183, 780, 1049, 1992, 957, 9
91, 538, 1434, 1272, 1103, 1356, 889, 1200, 933, 830, 564, 1046, 1767, 1027,
831, 1720, 1061, 1344, 687, 2122, 990, 727, 1121, 1139, 728, 1317, 1014, 129
0, 1439, 1426, 1019, 1065, 1102, 982, 880, 1341, 2029, 1499, 839, 1399, 364
3, 794, 684, 906, 1294, 965, 970, 1973, 1389, 666, 1164, 595, 1081, 1180, 16
49, 1310, 1397, 1153, 1130, 1669, 1071, 1056, 778, 808, 1615, 981, 1228, 77
9, 1424, 1142, 1412, 715, 1806, 1150, 873, 1432, 1339, 1473, 1275, 1008, 158
4, 1448, 1151, 1390, 1319, 1175, 1673, 903, 978, 1503, 1127, 909, 1360, 137
3, 913, 1425, 1105, 1039, 848, 1877, 1342, 1134, 908, 898, 1352, 1545, 1833,
1220, 1281, 1029, 1508, 661, 608, 1206, 1237, 1287, 1148, 1037, 1226, 1608,
1176, 1153, 874, 1960, 1331, 1776, 1327, 1090, 2078, 1124, 1309, 1300, 1359,

1204, 1756, 1441, 1491, 915, 1869, 823, 1850, 1820, 1602, 1492, 1946, 743, 2015, 1031, 1368, 946, 1369, 1169, 1096, 499, 1076, 1569, 2516, 797, 1186, 1058, 1140, 1078, 1211, 2692, 921, 1739, 846, 1488, 976, 1311, 1332, 1227, 1197, 996, 1357, 2191, 1552, 1250, 1471, 1456, 1462, 1834, 3422, 876, 1934, 1306, 994, 765, 1070, 917, 1099, 1504, 1282, 971, 1181, 1133, 696, 1092, 1613, 1735, 1476, 1842, 969, 737, 1038, 1217, 3322, 1184, 1579, 1871, 1126, 1364, 1559, 1293, 1241, 1740, 975, 1464, 4199, 1505, 1532, 1117, 1123, 1258, 1329, 1809, 1366, 961, 1374, 1223, 2089, 1033, 1129, 1642, 2372, 3253, 2429, 3284, 1239, 4918, 1303, 1606, 1091, 960, 1978, 1122, 1855, 2, 1437, 1788, 972, 1554, 925, 998, 1057, 1548, 1283, 1502, 1265, 1662, 1468, 1337, 1435, 997, 1192, 1816, 1040, 1256, 931, 1490, 1154, 1035, 888, 1166, 1145, 2420, 1598, 2453, 1221, 1182, 1480, 980, 1571, 1555, 1067, 1447, 3076, 1530, 2870, 2316, 1353, 2299, 1, 2129, 1190, 1665, 1594, 1372, 1214, 1174, 1463, 1365, 1285, 1410, 1095, 840, 1624, 949, 2926, 2053, 2139, 1202, 1112, 1550, 1010, 1817, 1333, 1348, 1260, 1958, 1370, 841, 1240, 1100, 954, 1079, 1131, 1489, 1347, 1957, 1191, 1391, 877, 1262, 1954, 1024, 1514, 934, 1232, 1388, 1531, 1925, 1710, 1108, 1512, 1966, 937, 1970, 1302, 1975, 1805, 1279, 1020, 1723, 1543, 959, 1286, 1380, 1326, 2301, 1880, 3509, 1460, 1048, 2219, 1361, 1603, 883, 1225, 1416, 1398, 928, 2055, 1962, 1104, 1551, 1580, 1745, 2187, 1707, 1233, 2184, 1628, 1804, 2062, 1472, 2486, 1267, 1563, 1407, 2035, 3785, 1440, 1394, 1405, 1640, 1616, 1246, 1556, 1868]

campaign: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 19, 18, 23, 14, 22, 25, 16, 17, 15, 20, 56, 39, 35, 42, 28, 26, 27, 32, 21, 24, 29, 31, 30, 41, 37, 40, 33, 34, 43]

pdays: [-1, 6, 4, 3, 5, 1, 0, 10, 7, 8, 9, 11, 2, 12, 13, 14, 15, 16, 21, 17, 18, 22, 25, 26, 19, 27, 20]

previous: [0, 1, 2, 3, 4, 5, 6, 7]

poutcome: ['nonexistent', 'failure', 'success']

empvarrate: [1.1, 1.4, -0.1, -0.2, -1.8, -2.9, -3.4, -3.0, -1.7, -1.1]

conspriceidx: [93.994, 94.465, 93.918, 93.444, 93.798, 93.2, 92.756, 92.843, 93.075, 92.893, 92.963, 92.469, 92.201, 92.379, 92.431, 92.649, 92.713, 93.369, 93.749, 93.876, 94.055, 94.215, 94.027, 94.199, 94.601, 94.767]

consconfidx: [-36.4, -41.8, -42.7, -36.1, -40.4, -42.0, -45.9, -50.0, -47.1, -46.2, -40.8, -33.6, -31.4, -29.8, -26.9, -30.1, -33.0, -34.8, -34.6, -40.0, -39.8, -40.3, -38.3, -37.5, -49.5, -50.8]

euribor3m: [4.857, 4.856, 4.855, 4.859, 4.86, 4.858, 4.864, 4.865, 4.866, 4.967, 4.961, 4.959, 4.958, 4.96, 4.962, 4.955, 4.947, 4.956, 4.966, 4.963, 4.957, 4.968, 4.97, 4.965, 4.964, 5.045, 5.0, 4.936, 4.921, 4.918, 4.912, 4.827, 4.794, 4.76, 4.733, 4.7, 4.663, 4.592, 4.474, 4.406, 4.343, 4.286, 4.245, 4.223, 4.191, 4.153, 4.12, 4.076, 4.021, 3.901, 3.879, 3.853, 3.816, 3.743, 3.669, 3.563, 3.488, 3.428, 3.329, 3.282, 3.053, 1.811, 1.799, 1.778, 1.757, 1.726, 1.703, 1.687, 1.663, 1.65, 1.64, 1.629, 1.614, 1.602, 1.584, 1.574, 1.56, 1.556, 1.548, 1.538, 1.531, 1.52, 1.51, 1.498, 1.483, 1.479, 1.466, 1.453, 1.445, 1.435, 1.423, 1.415, 1.41, 1.405, 1.406, 1.4, 1.392, 1.384, 1.372, 1.365, 1.354, 1.344, 1.334, 1.327, 1.313, 1.299, 1.291, 1.281, 1.266, 1.25, 1.244, 1.259, 1.264, 1.27, 1.262, 1.26, 1.268, 1.286, 1.252, 1.235, 1.224, 1.215, 1.206, 1.099, 1.085, 1.072, 1.059, 1.048, 1.044, 1.029, 1.018, 1.007, 0.996, 0.979, 0.969, 0.944, 0.937, 0.933, 0.927, 0.921, 0.914, 0.908, 0.903, 0.899, 0.884, 0.883, 0.881, 0.879, 0.873, 0.869, 0.861, 0.859, 0.854, 0.851, 0.849, 0.843, 0.838, 0.834, 0.829, 0.825, 0.821, 0.819, 0.813, 0.809, 0.803, 0.797, 0.788, 0.781, 0.778, 0.773, 0.771, 0.77, 0.768, 0.766, 0.762, 0.755, 0.749, 0.743, 0.741, 0.739, 0.75, 0.753, 0.754, 0.752, 0.744, 0.74, 0.742, 0.737, 0.735, 0.733, 0.73, 0.731, 0.728, 0.724, 0.722, 0.72, 0.719, 0.716, 0.715, 0.714, 0.718, 0.721, 0.717, 0.712, 0.71, 0.709, 0.708, 0.706, 0.707, 0.7, 0.655, 0.654, 0.653, 0.652, 0.651, 0.65, 0.649, 0.646, 0.644, 0.643, 0.639, 0.637, 0.635, 0.636, 0.634, 0.638, 0.64, 0.642, 0.645, 0.659, 0.

```

663, 0.668, 0.672, 0.677, 0.682, 0.683, 0.684, 0.685, 0.688, 0.69, 0.692, 0.
695, 0.697, 0.699, 0.701, 0.702, 0.704, 0.711, 0.713, 0.723, 0.727, 0.729,
0.732, 0.748, 0.761, 0.767, 0.782, 0.79, 0.793, 0.802, 0.81, 0.822, 0.827,
0.835, 0.84, 0.846, 0.87, 0.876, 0.885, 0.889, 0.893, 0.896, 0.898, 0.9, 0.9
04, 0.905, 0.895, 0.894, 0.891, 0.89, 0.888, 0.886, 0.882, 0.88, 0.878, 0.87
7, 0.942, 0.953, 0.956, 0.959, 0.965, 0.972, 0.977, 0.982, 0.985, 0.987, 0.9
93, 1.0, 1.008, 1.016, 1.025, 1.032, 1.037, 1.043, 1.045, 1.047, 1.05, 1.04
9, 1.046, 1.041, 1.04, 1.039, 1.035, 1.03, 1.031, 1.028]
nremployed: [5191.0, 5228.1, 5195.8, 5176.3, 5099.1, 5076.2, 5017.5, 5023.5,
5008.7, 4991.6, 4963.6]
age_category: ['prime', 'young', 'veteran']
education_category: ['basic', 'higher', 'unknown', 'other']
duration_category: ['long', 'medium', 'short', 'very_long', 'very Short']
previous_contact: [0, 1]
campaign_category: ['low_engagement', 'moderate_engagement', 'high_engagemen
t']

```

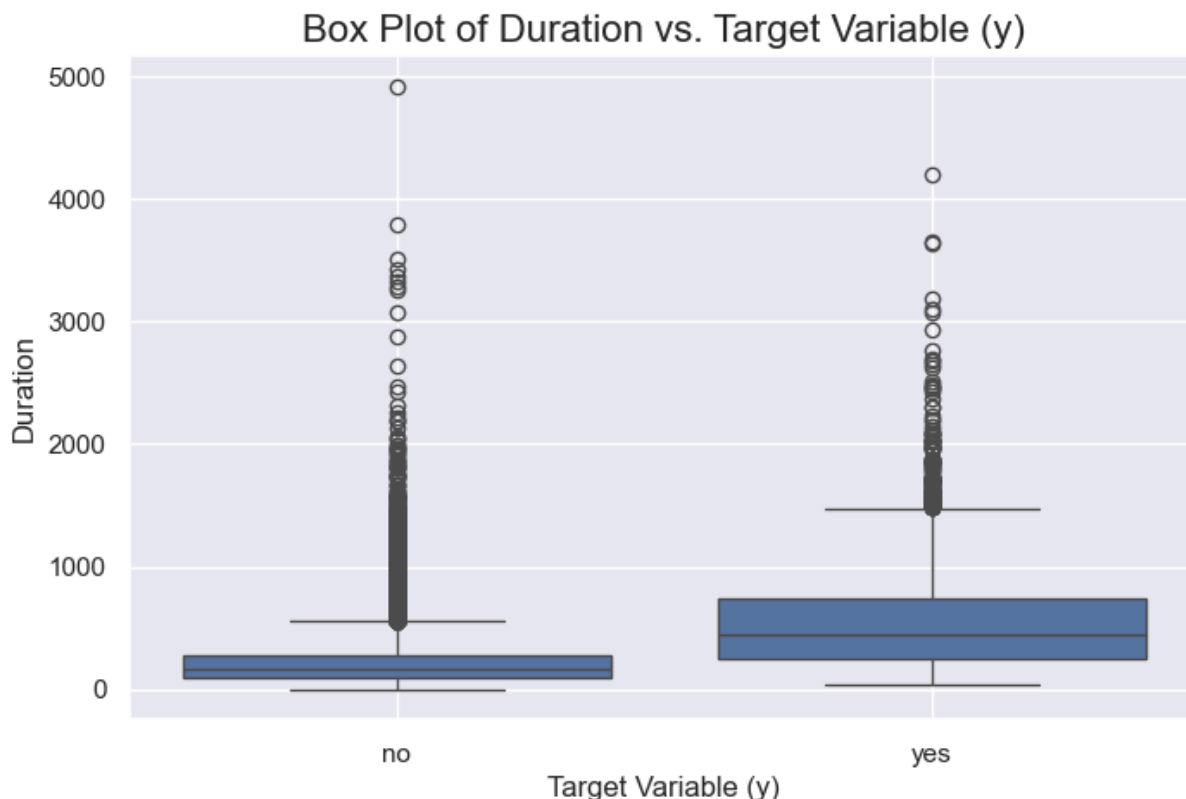
Data Analysis performance: Extracting insights and Trends in the data

duration

```

In [65]: # Creating a box plot for 'duration' by target variable 'y'
plt.figure(figsize=(8, 5)) # Set figure size for better visibility
sns.boxplot(data=df_processed, x='y', y='duration')
plt.title('Box Plot of Duration vs. Target Variable (y)', fontsize=16)
plt.xlabel('Target Variable (y)', fontsize=12)
plt.ylabel('Duration', fontsize=12)
# Displaying the plot
plt.grid(True) # Add grid for better readability
plt.show()

```



Observation:

The box plot reveals some outliers in the data, showing samples outside the range of statistical density. Additionally, there is a noticeable dependency: as the duration of contact with a client increases, the frequency of 'yes' decisions also rises. This suggests that longer contact durations may positively influence the likelihood of a favorable decision.

```
In [66]: for x in range(95, 101, 1):
          print("{}% of calls have duration less than equal to {}".format(x, df_pr
          iqr = df_processed.duration.quantile(0.75) - df_processed.duration.quantile(
          print('IQR {}'.format(iqr))
```

```
95% of calls have duration less than equal to 752.65000000000015
96% of calls have duration less than equal to 820.51999999999968
97% of calls have duration less than equal to 911.0
98% of calls have duration less than equal to 1052.2600000000002
99% of calls have duration less than equal to 1271.12999999999974
100% of calls have duration less than equal to 4918.0
IQR 217.0
```

Observation:

The results indicate that while 95% of calls have a duration of 752.65 seconds or less, with the maximum call lasting 4918 seconds, the interquartile range (IQR) of 217 seconds suggests a moderate variability in call durations among the central 50% of the dataset.

age

```
In [68]: plt.figure(figsize=(10, 6))
          # Create a box plot for 'age' by target variable 'y'
          sns.boxplot(data=df_processed, x='y', y='age')
          # Adding title and labels with improved font sizes
          plt.title('Box Plot of Age vs. Target Variable (y)', fontsize=16)
          plt.xlabel('Target Variable (y)', fontsize=12)
          plt.ylabel('Age', fontsize=12)
          # Add grid for better readability
          plt.grid(True)
          plt.show()
```

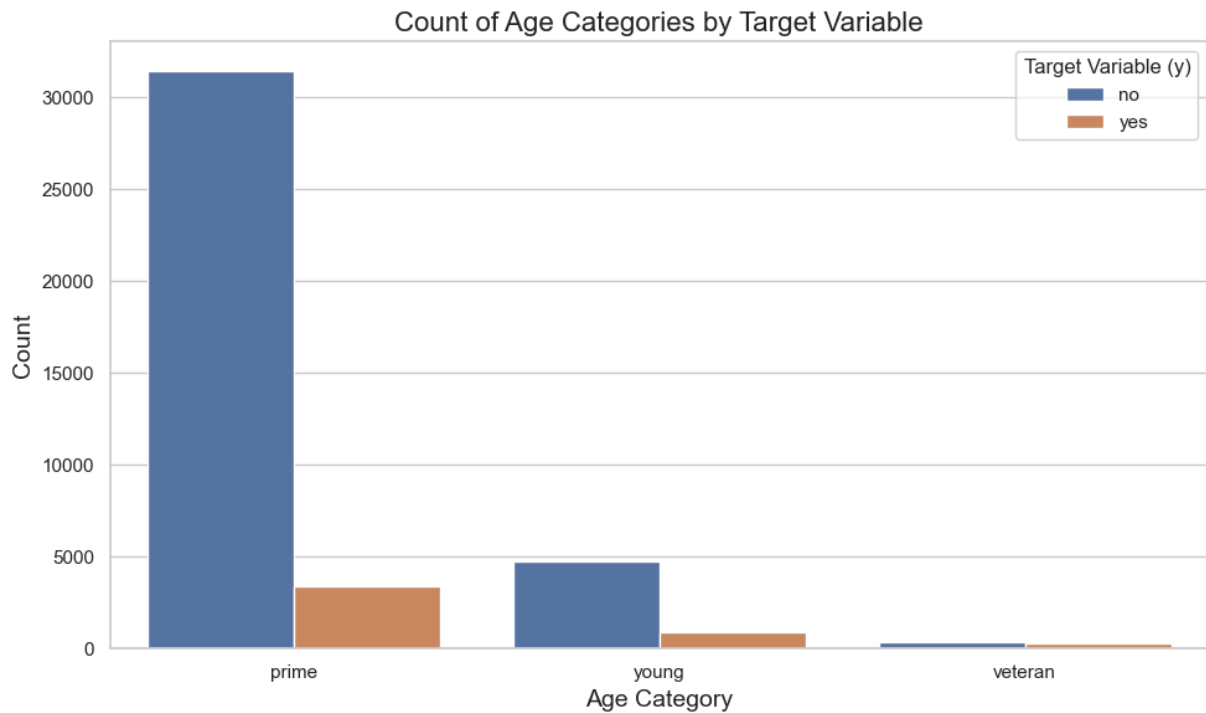


Observation:

There is no clear dependency on the year. The box plot shows the presence of outliers, but in both cases, the main data group is located in a similar age category, approximately between 30 and 48 years. However, individuals above and below this range are highly likely to fall into the 'yes' category. It is possible that age categories above and below the 30-48 range may have a slight impact on the final decision, though not significantly.

age category

```
In [72]: sns.set(style='whitegrid')
plt.figure(figsize=(10, 6))
sns.countplot(x='age_category', data=df_processed, hue='y')
plt.title('Count of Age Categories by Target Variable', fontsize=16)
plt.xlabel('Age Category', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.legend(title='Target Variable (y)', loc='upper right')
plt.tight_layout()
plt.show()
```

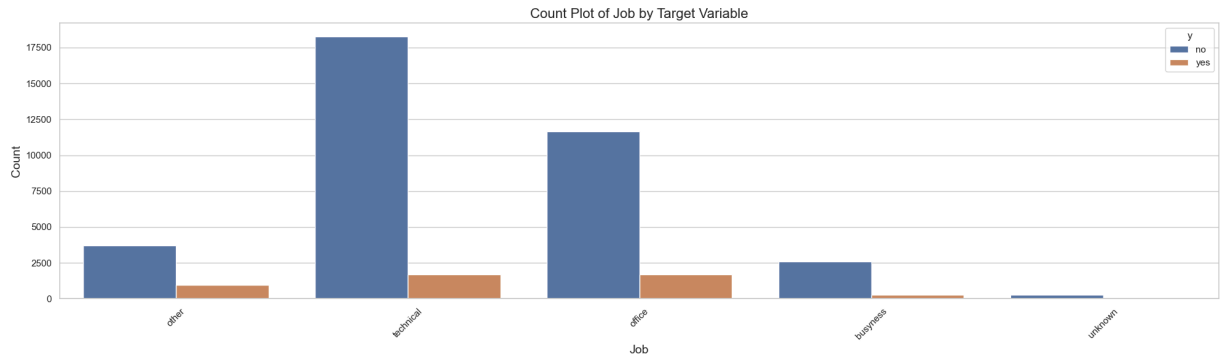
Observation:

There is a clear dependency within the age categories. The younger group tends to lean towards 'no' decisions, while the 'prime' age category shows a higher frequency of 'yes' decisions. The age category may have a significant impact on modeling, as older individuals (veterans) are more likely to agree.

Additionally, younger people are generally more willing to take risks and engage in speculative investments, whereas older individuals prefer more stable options, such as bank accounts or bonds. Furthermore, older people tend to be less open-minded than younger individuals, which increases the likelihood that an older person will decide 'yes.'

job

```
In [73]: plt.figure(figsize=(20, 6))
# Create a count plot with better handling of hue
sns.countplot(x='job', data=df_processed, hue='y')
# Set the title and labels
plt.title('Count Plot of Job by Target Variable', fontsize=16)
plt.xlabel('Job', fontsize=14)
plt.ylabel('Count', fontsize=14)
# Rotate x-tick labels for better visibility if needed
plt.xticks(rotation=45)
# Display the plot
plt.tight_layout()
plt.show()
```

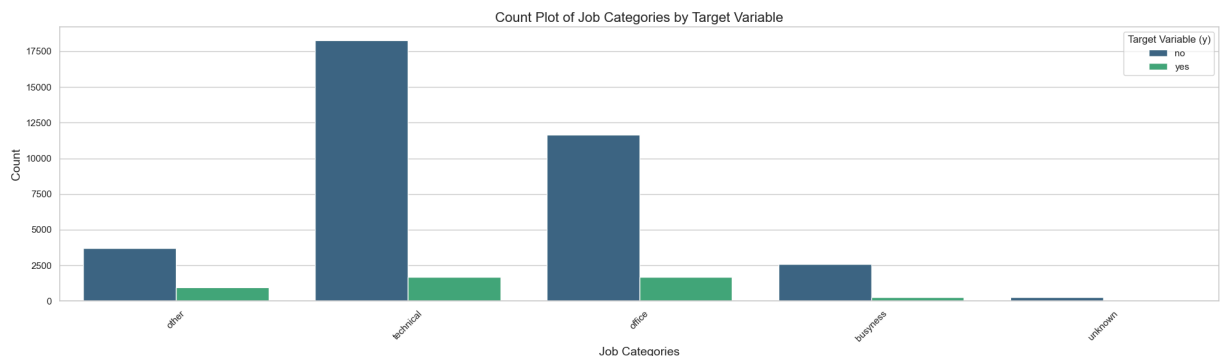


Observation:

The dominant job categories in the 'yes' group are "technical" and "office." However, when considering the ratio within each category, it is clear that the "other" category has the highest number of 'yes' samples relative to the total samples within that category. I suspect that the significant impact comes from students, which we will verify in the next plot. Therefore, I believe the job feature also has the potential to influence the outcome.

job categories

```
In [74]: plt.figure(figsize=(20, 6))
# Create a count plot for job categories with the target variable as hue
sns.countplot(x='job', data=df, hue='y', palette='viridis')
plt.title('Count Plot of Job Categories by Target Variable', fontsize=16)
plt.xlabel('Job Categories', fontsize=14)
plt.ylabel('Count', fontsize=14)
# Rotate x-axis labels for better readability
plt.xticks(rotation=45)
plt.legend(title='Target Variable (y)')
plt.tight_layout()
plt.show()
```



Observation:

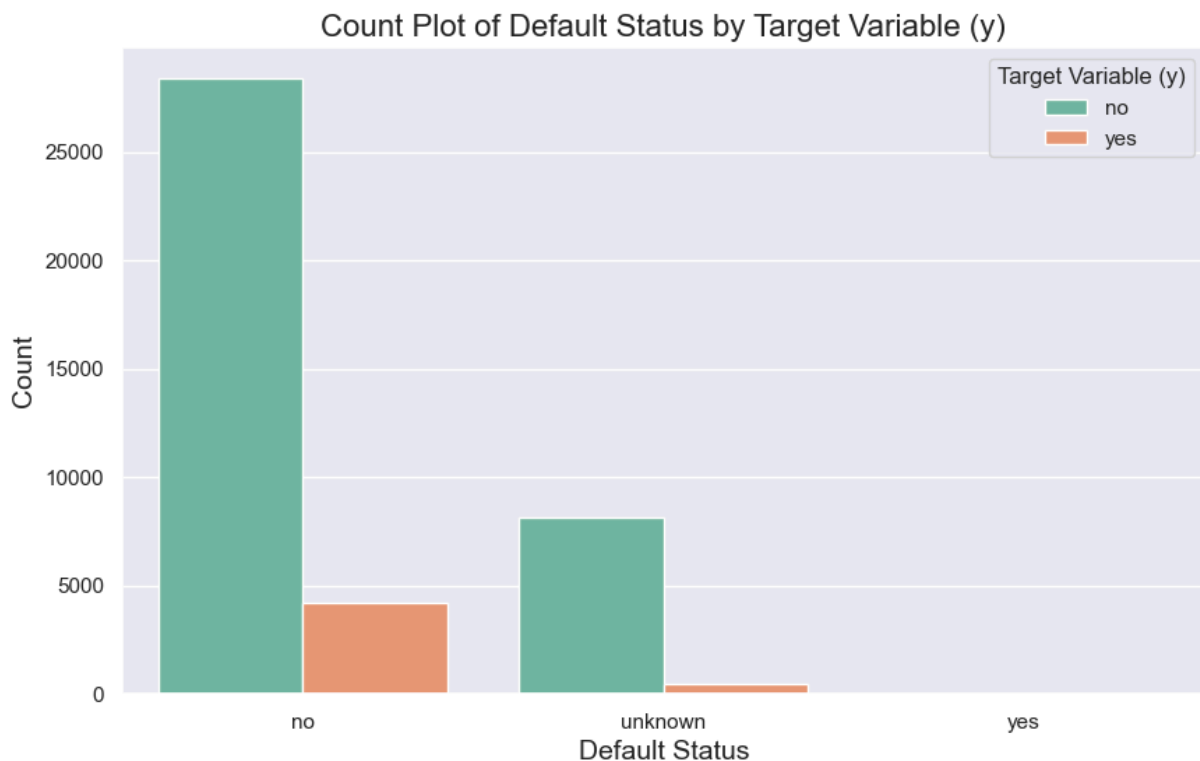
Considering the total number of 'yes' samples per job group, the leading category is 'admin.' However, this category also has the highest number of 'no' samples, so we must analyze the ratio of total samples within each group and the ratio between 'yes' and 'no' responses. Based on this analysis, the highest

proportion of 'yes' decisions belongs to the 'retired' and 'student' categories, which correlates with the visualization related to age categories. Conversely, the lowest ratios are observed in the 'unemployed,' 'self-employed,' and 'housemaid' categories.

From this observation, my theory is that this feature will significantly impact the outcomes. Moreover, from a business strategy perspective, the marketing team should focus its promotional efforts more on specific job categories.

default

```
In [75]: sns.set(rc={'figure.figsize': (10, 6)})
# Create count plot
sns.countplot(x='default', data=df_processed, hue='y', palette='Set2')
plt.title('Count Plot of Default Status by Target Variable (y)', fontsize=16)
plt.xlabel('Default Status', fontsize=14)
plt.ylabel('Count', fontsize=14)
# Show the plot
plt.legend(title='Target Variable (y)')
plt.show()
```



Observation:

Most people are not in default, and those free from financial responsibilities are more likely to decide 'yes,' as seen in the plot. Conversely, the 'unknown' category shows the lowest ratio of 'yes' decisions relative to the total number of people in that group. It can be inferred that individuals may be reluctant to disclose their financial status, and those in default are more likely to respond 'no'

or select 'unknown' compared to those without defaults. This behavior highlights the potential impact of financial status on decision-making.

loan

```
In [14]: df_processed.loan.value_counts()
```

```
Out[14]: loan
no          33950
yes          6248
unknown      990
Name: count, dtype: int64
```

```
In [77]: plt.figure(figsize=(10, 6))
# Create the count plot for the 'loan' feature
sns.countplot(x='loan', data=df_processed, hue='y', palette='Set2')
plt.title('Count Plot of Loan Status for Target Variable y', fontsize=16)
plt.xlabel('Loan Status', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.legend(title='Target Variable (y)', fontsize=12)
# Display the plot
plt.tight_layout()
plt.show()
```



Observation:

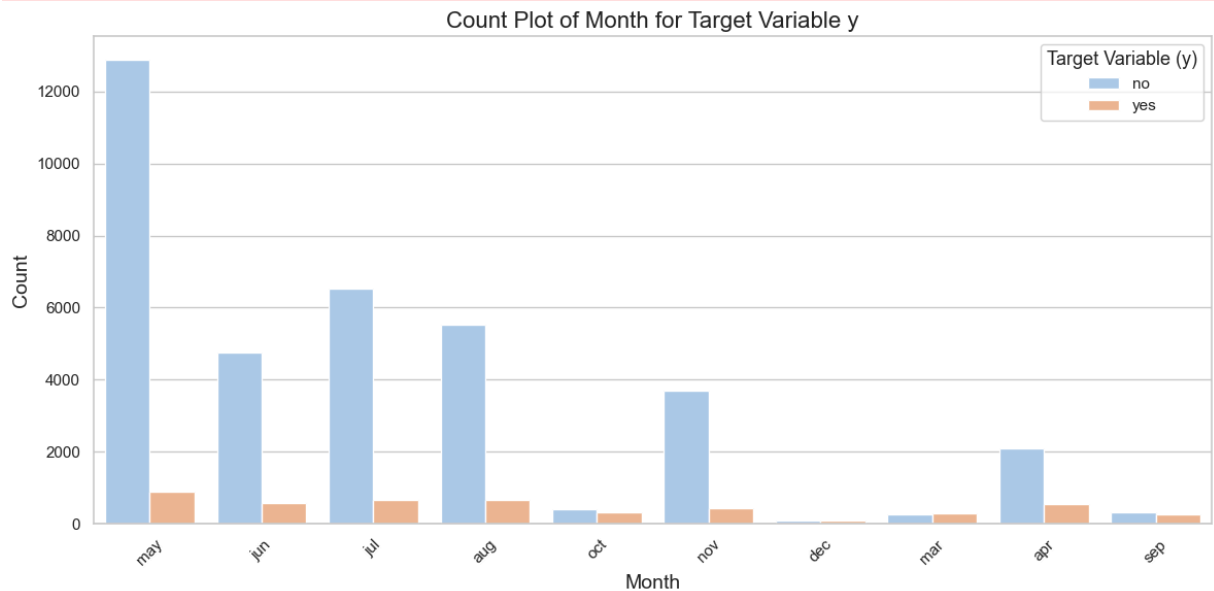
The same pattern and tendency are visible in the case of the 'loan' feature. Similar to the 'default' variable, people with fewer financial resources are less likely to make a 'yes' decision.

month

```
In [78]: sns.set(rc={'figure.figsize': (12, 6)})
sns.set_style('whitegrid')
# Create the count plot
sns.countplot(x='month', data=df_processed, hue='y', palette='pastel')
# Add title and labels
plt.title('Count Plot of Month for Target Variable y', fontsize=16)
plt.xlabel('Month', fontsize=14)
plt.ylabel('Count', fontsize=14)
# Improve x-tick rotation for better readability
plt.xticks(rotation=45)
plt.legend(title='Target Variable (y)', title_fontsize='13', fontsize='11')
# Display the plot
plt.tight_layout()
plt.show()
```

2024-10-05 13:47:58,082 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.

2024-10-05 13:47:58,121 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.



Observation:

The highest number of contacts per month is observed in May, while December has the lowest. A moderate number of contacts is recorded in June, July, and August. It raises a good question: what is the basis for this trend? October, December, March, and September show the lowest statistics. Why is May particularly significant? One possible explanation is that the summer quarter is a vacation period, making people more available for contact. However, what about November? Perhaps "Black Friday" plays a role. While these trends are visible, they suggest a potential but not strong impact on decision-making.

euribor3m

```
In [79]: plt.figure(figsize=(12, 6))
# Create the count plot with improved aesthetics
sns.countplot(x='euribor3m', data=df, hue='y', palette='viridis')
plt.title('Count Plot of Euribor 3 Month for Target Variable y', fontsize=16)
plt.xlabel('Euribor 3 Month', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.legend(title='Target Variable (y)', title_fontsize='13', fontsize='11')
# Display the plot
plt.tight_layout()
plt.show()
```

2024-10-05 13:49:51,717 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.
 2024-10-05 13:49:51,749 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.



Observation:

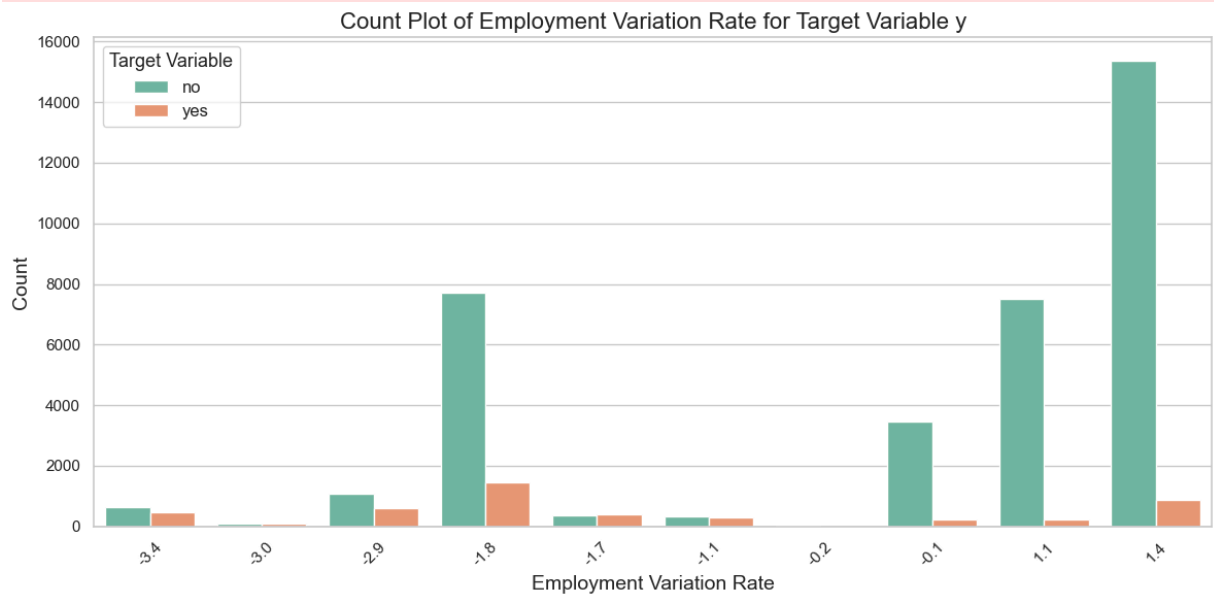
EURIBOR – (Euro Interbank Offered Rate) – is the interest rate (reference indicator) at which banks are willing to lend euros to other banks in the Eurozone interbank market. Certain values of EURIBOR show a strong decline toward 'yes' decisions, leading to the conclusion that this feature will definitely impact customer decisions. The business team must consider this macroeconomic statistic and engage customers more frequently than usual.

This time period is influenced by macroeconomic parameters that can increase the number of customers. Lower interest rates equate to "cheap" money, which means lower costs for credit. As a result, banks can propose more benefits to clients and vice versa.

empvarrate

```
In [80]: plt.figure(figsize=(12, 6))
# Create count plot for empvarrate with hue for target variable
sns.countplot(x='empvarrate', data=df, hue='y', palette='Set2')
plt.title('Count Plot of Employment Variation Rate for Target Variable y', f
plt.xlabel('Employment Variation Rate', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.xticks(rotation=45)
plt.legend(title='Target Variable', title_fontsize='13', fontsize='12')
# Show the plot
plt.tight_layout()
plt.show()
```

2024-10-05 13:55:42,940 - INFO - Using categorical units to plot a list of s
 trings that are all parsable as floats or dates. If these strings should be
 plotted as numbers, cast to the appropriate data type before plotting.
 2024-10-05 13:55:42,971 - INFO - Using categorical units to plot a list of s
 trings that are all parsable as floats or dates. If these strings should be
 plotted as numbers, cast to the appropriate data type before plotting.



Observation:

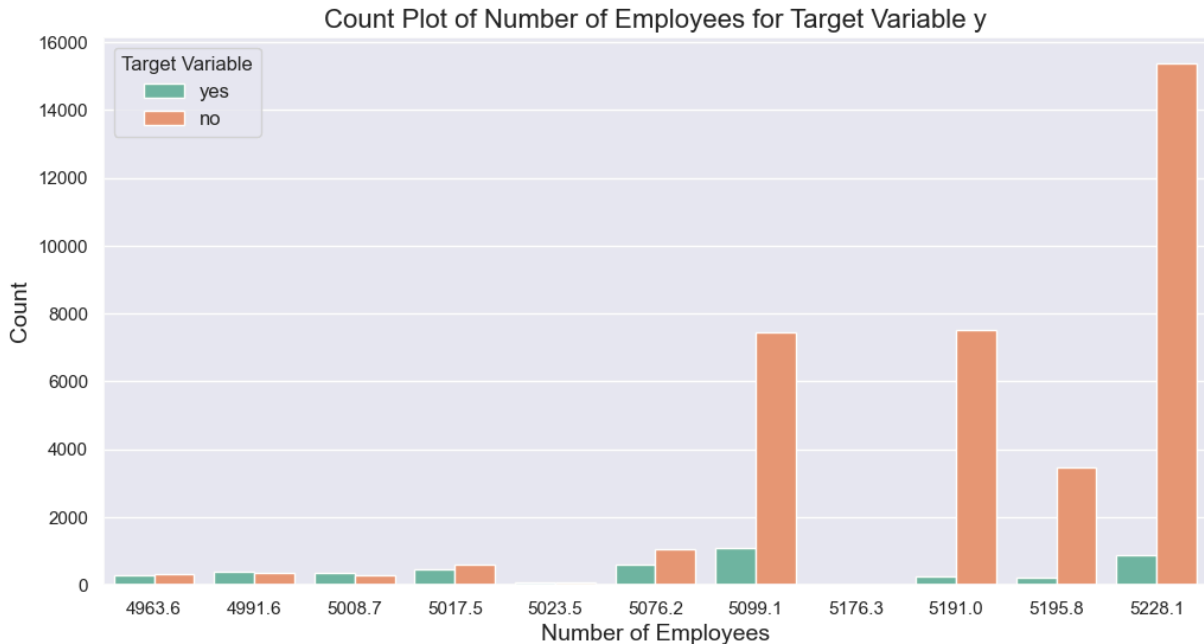
A similar trend is observed in the case of EURIBOR, with notable peaks at values of -1.8 and 1.4, which warrant further investigation. Additional macroeconomic parameters exhibiting a strong decision-making pattern should be considered by the marketing business team. It might be beneficial to propose more incentives to customers during these periods.

nremployed

```
In [81]: sns.set(rc={'figure.figsize': (12, 6)})
sns.countplot(x='nremployed', data=df, hue='y', palette='Set2')
plt.title('Count Plot of Number of Employees for Target Variable y', fontsize=14)
plt.xlabel('Number of Employees', fontsize=14)
plt.ylabel('Count', fontsize=14)
```

```
plt.legend(title='Target Variable', fontsize=12)
plt.show()
```

2024-10-05 13:57:25,605 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.
 2024-10-05 13:57:25,651 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.



Observation:

Such macroeconomic parameters should be considered as well, as we can observe a strong inclination towards 'no' decisions with an increase in the total number of employed people. While there is no strong recommendation, these anomalies are a good point for further investigation. More employed people typically indicate a strong macroeconomic state, which, in turn, means lower interest rates on deposits. Consequently, people earn less from saving money in banks and are more likely to invest in riskier activities. This tendency may explain the trend of rising 'no' decisions alongside an increase in the total number of employed individuals. This parameter should also be considered by the marketing team in their business strategy.

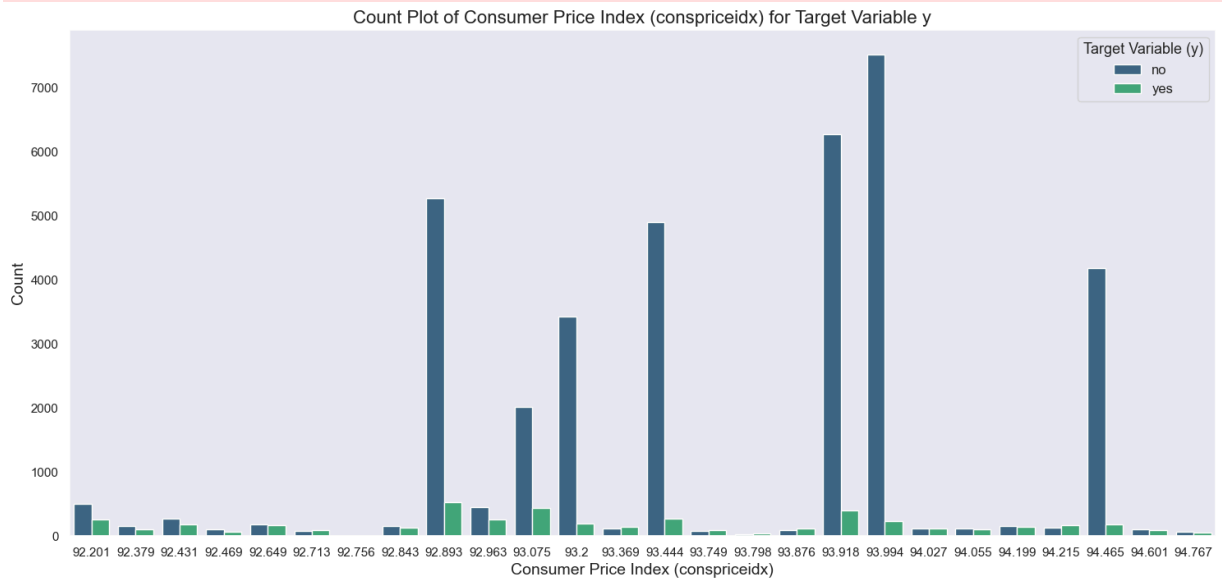
conspiceidx

```
In [91]: sns.set(rc={'figure.figsize': (18,8)})
# Create the count plot
sns.countplot(x='conspiceidx', data=df, hue='y', palette='viridis')
plt.title('Count Plot of Consumer Price Index (conspiceidx) for Target Vari
plt.xlabel('Consumer Price Index (conspiceidx)', fontsize=14)
plt.ylabel('Count', fontsize=14)
# Show the plot
plt.show()
```



```
plt.grid(axis='y')
plt.show()
```

2024-10-05 14:01:02,156 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.
 2024-10-05 14:01:02,198 - INFO - Using categorical units to plot a list of strings that are all parsable as floats or dates. If these strings should be plotted as numbers, cast to the appropriate data type before plotting.



Observation:

The Consumer Confidence Index (CCI) measures the degree of optimism consumers have regarding current and expected economic conditions. There are numerous peaks visible, which indicate that more contacts were made during these periods, as shown in the monthly contact data.

However, there is something underlying these trends that requires further investigation, as additional data is needed for a more precise analysis. Some periods show strong declines, during which people were more likely to make 'no' decisions or reject offers from banks.

It would be beneficial to conduct a deeper analysis to determine the specific time frames when these declines occurred and to identify the events or sentiments that dominated during those times. This feature holds significant potential for influencing decision-making and should be considered by the marketing team in their business strategy.

Recommendations and conclusion:

- Personal statistics are important, such as the **duration of contact**, the financial state of the individual or family, and the timing of the last contact. These factors have a strong potential to influence decision-making and **should be considered within the marketing strategy as well.**

- Based on the observations and data analysis, a strong potential impact of macroeconomic statistics has been identified. This impact must be considered by the BAU team to enhance their marketing strategy in light of the provided insights. **However, a deeper analysis is necessary to obtain a clearer overview of macroeconomic statistics, as there is also a tendency for an increased number of contacts with customers. Additionally, I would like to analyze time series data for better insight into the situation. For now, I cannot conclude that these parameters should be dropped from consideration, as they have a high potential to impact the model and may lead to more false positive decisions.**

The AutoEDA class was developed to extract essential information for model parameterization and selection, including missing values, balance in label distribution, and other key metrics. This functionality helps streamline the exploratory data analysis process, providing insights necessary for effective modeling. The class aims to simplify data preparation and enhance the overall efficiency of the modeling workflow.

```
In [21]: class AutoEDA:
    def __init__(self, df):
        """initialize the class with a DataFrame."""
        self.df = df
        self._encode_categorical_columns()
    def _encode_categorical_columns(self):
        """automatically encode categorical columns to numeric using Label Encoder"""
        categorical_cols = self.df.select_dtypes(include=['object']).columns
        self.label_encoders = {}
        for col in categorical_cols:
            le = LabelEncoder()
            self.df[col] = le.fit_transform(self.df[col])
            self.label_encoders[col] = le # Store the encoder for future reference
    def missing_values(self):
        """display missing values count and percentage."""
        missing_data = self.df.isnull().sum()
        missing_percentage = (self.df.isnull().mean() * 100).round(2)
        missing_df = pd.DataFrame({'Missing Values': missing_data, 'Percentage': missing_percentage})
        return missing_df[missing_df['Missing Values'] > 0].sort_values(by='Missing Values', ascending=False)
    def display_missing_vals(self):
        """plot the heatmap with missing values"""
        sns.heatmap(df.isnull(), cbar=False, cmap='viridis')
        plt.title('Missing Data Heatmap')
        return plt.show()
    def summary_stats(self):
        """show summary statistics of numeric columns."""
        return self.df.describe(include = 'all').T
    def correlation_matrix(self):
        """plot a correlation heatmap."""
```

```

plt.figure(figsize=(15, 9))
sns.heatmap(self.df.corr(), annot=True, cmap='coolwarm', fmt='.2f',
plt.title('Correlation Matrix')
plt.show()
def target_correlation(self, target):
    """show correlation of all variables with the target variable."""
    correlations = self.df.corr()[target].sort_values(ascending=False)
    return correlations
def target_corr_stage(self, target):
    """show the stage of feature corr in relation to target"""
    le = LabelEncoder()
    df[target] = le.fit_transform(df[target])
    # Convert all other categorical columns to numeric (if needed)
    df_numeric = df.select_dtypes(include=[int, float])
    # Compute correlation matrix
    corr = df_numeric.corr()
    # Extract correlations with the target variable
    target_corr = corr[[target]].sort_values(by=target, ascending=False)
    # Create a heatmap of correlations
    plt.figure(figsize=(10, 6))
    sns.heatmap(target_corr, annot=True, cmap='coolwarm', vmin=-1, vmax=
plt.title(f'Correlation with {target}')
    return plt.show()
def outlier_detection(self):
    """detect outliers using box plots."""
    numeric_cols = self.df.select_dtypes(include=['float64', 'int64']).c
    plt.figure(figsize=(12, 6))
    for i, col in enumerate(numeric_cols, 1):
        plt.subplot(len(numeric_cols)//3+1, 3, i)
        sns.boxplot(x=self.df[col])
        plt.title(f'Outlier Detection: {col}')
    plt.tight_layout()
    plt.show()
def balance_in_target(self, target):
    """check the balance of classes in the target column."""
    target_counts = self.df[target].value_counts()
    total = len(self.df)
    for label, count in target_counts.items():
        print(f"Class {label}: {count} instances, {count / total * 100:.
    plt.figure(figsize=(6, 4))
    sns.barplot(x=target_counts.index, y=target_counts.values)
    plt.title(f'Class Distribution in {target}')
    plt.xlabel('Class')
    plt.ylabel('Count')
    plt.show()
def run_eda(self, target=None):
    """run all EDA steps automatically."""
    print("Missing Values:")
    print(self.missing_values())
    self.display_missing_vals()
    print("\nSummary Statistics:")
    print(self.summary_stats())
    self.correlation_matrix()
    self.target_corr_stage(target)
    if target:
        print(f"\nCorrelation with Target ({target}):")

```

```

        print(self.target_correlation(target))
        print("\nClass Balance Check:")
        self.balance_in_target(target)
        self.outlier_detection()

```

```

eda = AutoEDA(df)
eda.run_eda(target='y')

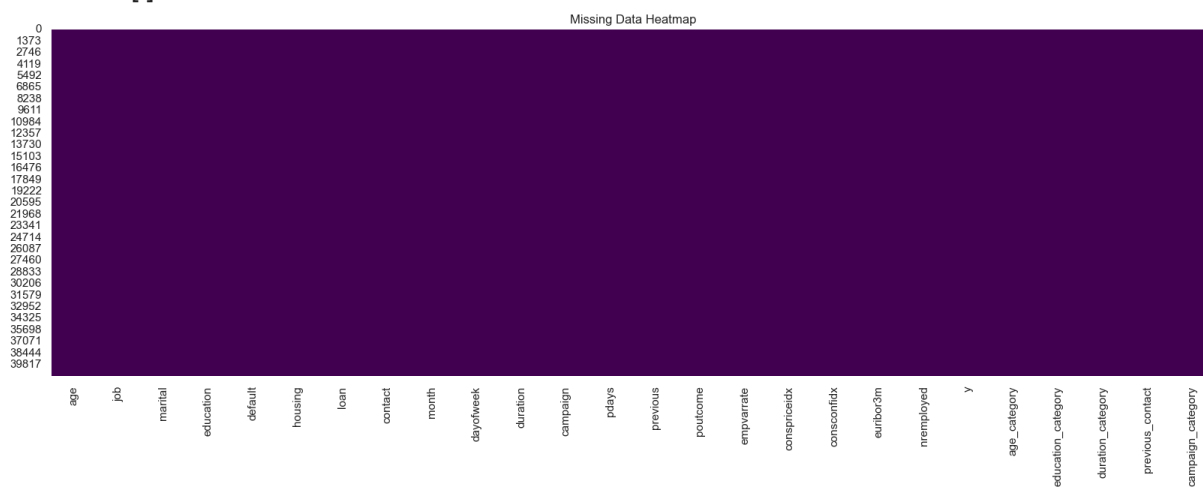
```

Missing Values:

Empty DataFrame

Columns: [Missing Values, Percentage]

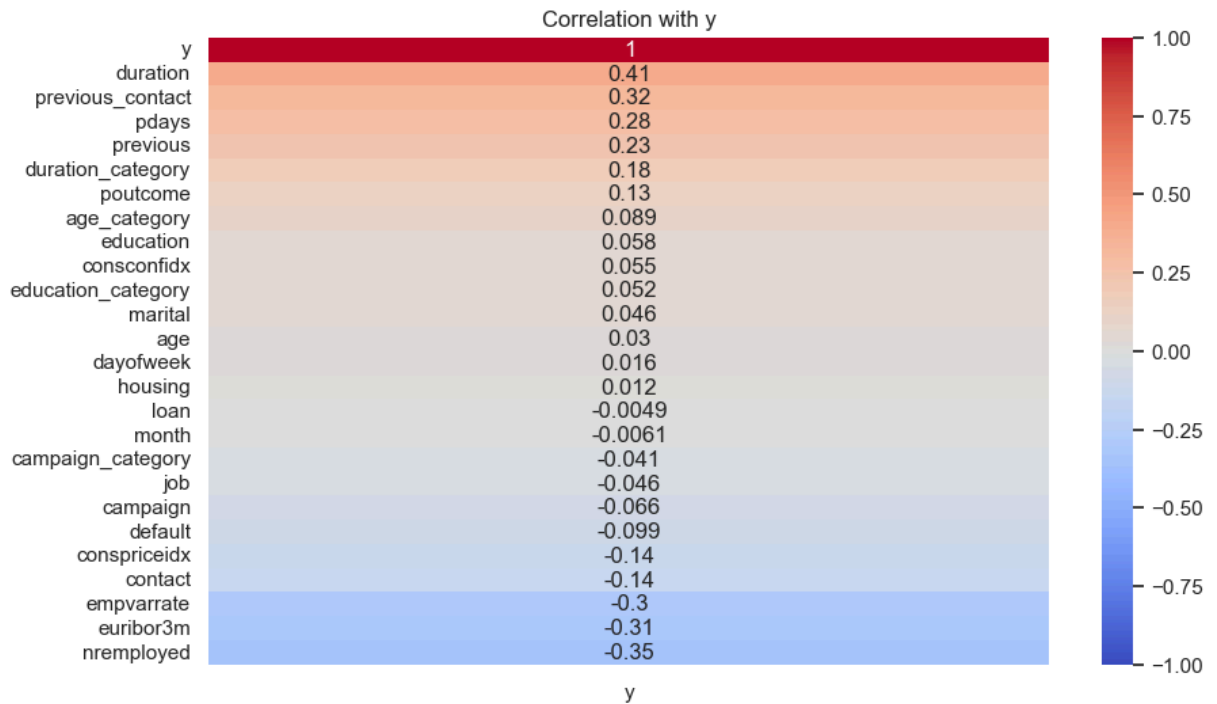
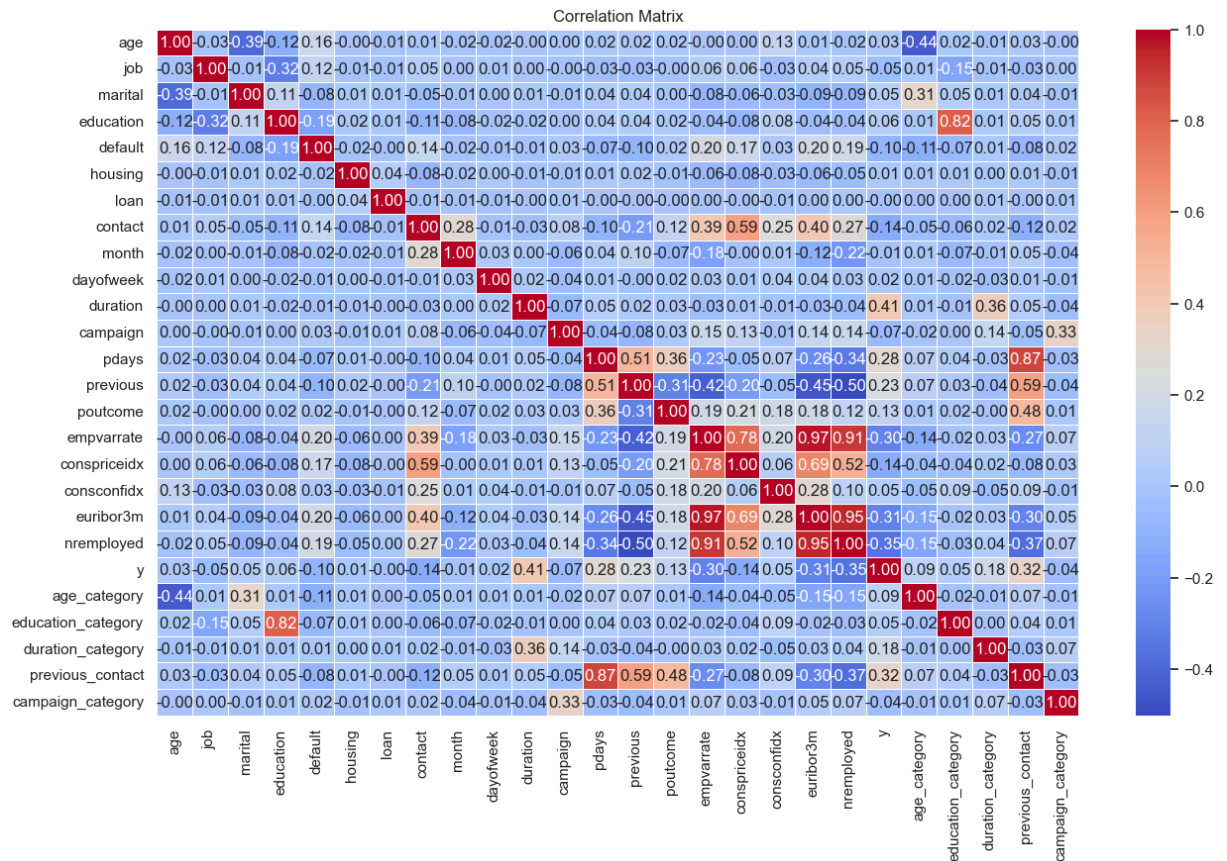
Index: []



Summary Statistics:

	count	mean	std	min	25%	\
age	41188.0	40.024060	10.421250	17.000	32.000	
job	41188.0	2.037050	1.057773	0.000	1.000	
marital	41188.0	1.172769	0.608902	0.000	1.000	
education	41188.0	3.747184	2.136482	0.000	2.000	
default	41188.0	0.208872	0.406686	0.000	0.000	
housing	41188.0	1.071720	0.985314	0.000	0.000	
loan	41188.0	0.327425	0.723616	0.000	0.000	
contact	41188.0	0.365252	0.481507	0.000	0.000	
month	41188.0	4.230868	2.320025	0.000	3.000	
dayofweek	41188.0	2.004613	1.397575	0.000	1.000	
duration	41188.0	258.285010	259.279249	0.000	102.000	
campaign	41188.0	2.567593	2.770014	1.000	1.000	
pdays	41188.0	-0.741988	1.510327	-1.000	-1.000	
previous	41188.0	0.172963	0.494901	0.000	0.000	
poutcome	41188.0	0.930101	0.362886	0.000	1.000	
empvarrate	41188.0	0.081886	1.570960	-3.400	-1.800	
conspriceidx	41188.0	93.575664	0.578840	92.201	93.075	
consconfidx	41188.0	-40.502600	4.628198	-50.800	-42.700	
euribor3m	41188.0	3.621291	1.734447	0.634	1.344	
nremployed	41188.0	5167.035911	72.251528	4963.600	5099.100	
y	41188.0	0.112654	0.316173	0.000	0.000	
age_category	41188.0	0.291371	0.694089	0.000	0.000	
education_category	41188.0	0.549675	0.707516	0.000	0.000	
duration_category	41188.0	0.985020	1.183031	0.000	0.000	
previous_contact	41188.0	0.036783	0.188230	0.000	0.000	
campaign_category	41188.0	1.101292	0.332789	0.000	1.000	

	50%	75%	max
age	38.000	47.000	98.000
job	2.000	3.000	4.000
marital	1.000	2.000	3.000
education	3.000	6.000	7.000
default	0.000	0.000	2.000
housing	2.000	2.000	2.000
loan	0.000	0.000	2.000
contact	0.000	1.000	1.000
month	4.000	6.000	9.000
dayofweek	2.000	3.000	4.000
duration	180.000	319.000	4918.000
campaign	2.000	3.000	56.000
pdays	-1.000	-1.000	27.000
previous	0.000	0.000	7.000
poutcome	1.000	1.000	2.000
empvarrate	1.100	1.400	1.400
conspriceidx	93.749	93.994	94.767
consconfidx	-41.800	-36.400	-26.900
euribor3m	4.857	4.961	5.045
nremployed	5191.000	5228.100	5228.100
y	0.000	0.000	1.000
age_category	0.000	0.000	2.000
education_category	0.000	1.000	3.000
duration_category	1.000	1.000	4.000
previous_contact	0.000	0.000	1.000
campaign_category	1.000	1.000	2.000



Correlation with Target (y):

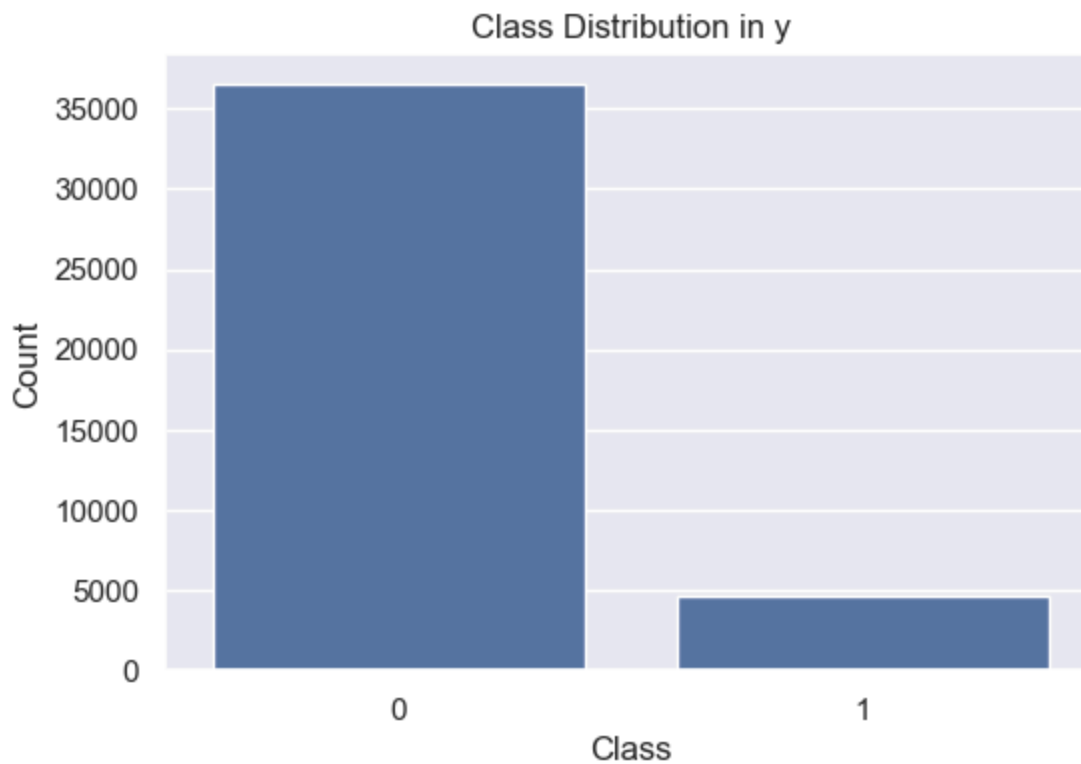
y	1.000000
duration	0.405274
previous_contact	0.324877
pdays	0.279025
previous	0.230181
duration_category	0.177240
poutcome	0.129789
age_category	0.089067
education	0.057799
consconfidx	0.054878
education_category	0.051610
marital	0.046203
age	0.030399
dayofweek	0.015967
housing	0.011552
loan	-0.004909
month	-0.006065
campaign_category	-0.041072
job	-0.045948
campaign	-0.066357
default	-0.099352
conspriceidx	-0.136211
contact	-0.144773
empvarrate	-0.298334
euribor3m	-0.307771
nremployed	-0.354678

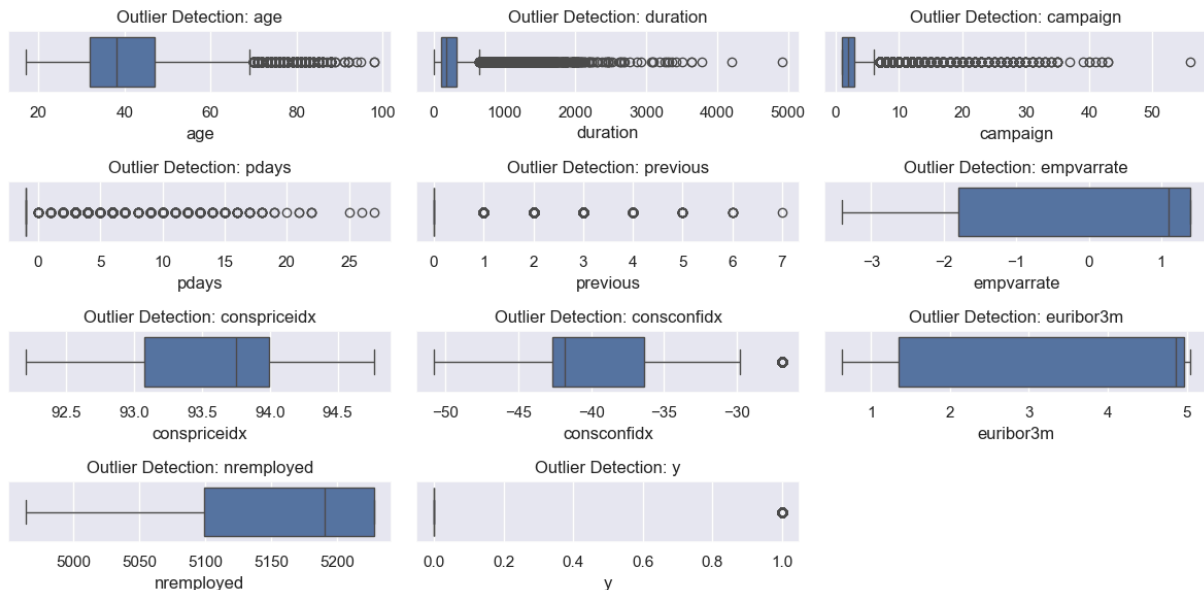
Name: y, dtype: float64

Class Balance Check:

Class 0: 36548 instances, 88.73% of total

Class 1: 4640 instances, 11.27% of total





Key statistics

missing values and duplicates:

- No missing values detected.
- Duplicates present but not a concern due to similar categories/encoding.

age:

- Mean: **40.11** years (SD: **10.31**)
- Range: **18** to **88** years
- Percentiles: 25th - **32**, Median - **38**, 75th - **47** **job:**
- Mean encoding: **3.82** (SD: **3.61**)
- Range: **0** to **11** (75th percentile: **7**) **marital status:**
- Mean: **1.18** (SD: **0.61**)
- Range: **0** to **3** (likely indicating 'married') **education:**
- Mean: **3.78** (SD: **2.15**)
- Range: **0** to **7** (75th percentile: **6**) **default:**
- Mean default rate: **0.195** (19.5% defaulted, SD: **0.397**) **housing:**
- Mean: **1.08** (indicates many clients have housing loans) **loan:**
- Mean: **0.35** (about **35%** have personal loans) **contact:**
- Mean encoding: **0.36** (binary contact types, max: **1**) **month:**
- Average contact month: **4.29** (indicating mid-year frequency) **day of week:**
- Average: **2.01** (encoded from **0** to **4**) **duration:**
- Mean call duration: **256.79** seconds (SD: **254.70**)
- 75th percentile: **317** seconds (max: **3643** seconds) **campaign:**
- Mean contacts during the campaign: **2.54** (up to **35** contacts) **pdays:**
- Majority have not been contacted in previous campaigns (**999**) **previous:**
- Mean previous contacts: **0.19** (most not contacted) **poutcome:**
- Mean outcome: **0.92** (mostly unsuccessful)

economic indicators:

- **Employment Variation Rate:** Mean **0.085** (slight positive variation)
- **Consumer Price Index:** Mean **93.58**
- **Consumer Confidence Index:** Mean **-40.50** (negative sentiment)
- **Euribor 3M:** Mean **3.62**
- **Number of Employees:** Mean **5166.48**

target variable (y)

- Mean: **0.109** (around **10.9%** subscribed to term deposits)

insights:

- Dataset contains a mix of categorical and numeric variables.
- Most clients did not default, take loans, or subscribe.
- High variance in **duration** and **campaign** indicates diverse interactions.
- Economic indicators suggest negative sentiment during the campaign.

correlation with target (y)

positive correlations

- **Duration: 0.419** (longer calls = higher subscription likelihood)
- **Previous: 0.256** (more previous contacts = slightly higher likelihood)
- **Poutcome: 0.123** (mildly positive)
- **Education: 0.067**
- **Age: 0.060**
- **Consumer Confidence Index: 0.054**
- **Marital Status: 0.044**
- **Job: 0.027**
- **Month: 0.005**
- **Housing: 0.001**

negative correlations

- **Day of Week: -0.006**
- **Loan: -0.013**
- **Campaign: -0.076** (more contacts = lower likelihood)
- **Default: -0.077**
- **Consumer Price Index: -0.098**
- **Contact: -0.137**
- **Employment Variation Rate: -0.283**
- **Euribor 3M: -0.299**
- **Pdays: -0.332**
- **Number of Employees: -0.349**

further insights

- The target variable "y" is imbalanced; consider techniques like SMOTE or class weight adjustments.
- **Duration** is the most significant positive factor for subscription.
- Economic factors (**Employment Variation Rate, Euribor 3M, pdays, Number of Employees**) strongly influence subscription likelihood.
- Demographic factors (**age, education, marital status**) show weaker predictive power compared to call-related factors and economic indicators.

experimental evaluation preprocessing

Feature selection techniques + SMOTE In this project, feature selection optimizes model performance by identifying impactful features and eliminating irrelevant ones. This enhances efficiency and accuracy. Following feature selection, SMOTE (Synthetic Minority Over-sampling Technique) addresses class imbalance by generating synthetic examples of the minority class, ensuring balanced training data.

Recursive Feature Elimination (RFE): RFE systematically ranks and selects the best subset of features by fitting a machine learning model and recursively removing the least important ones. This process continues until only the most relevant features remain, improving the model's predictive power and reducing overfitting.

The script preprocesses a dataset by separating features (X) and the target variable (y), applying SMOTE to address class imbalance, and scaling numerical features using StandardScaler. It then splits the dataset into training and testing sets before defining several estimators, including Decision Tree, Random Forest, Gradient Boosting, and XGBoost classifiers, and performs feature selection using Recursive Feature Elimination (RFE) for each model, printing the selected features. The script further counts and visualizes the frequency of selected features across models, retaining the top features for subsequent modeling, after which it trains each estimator and evaluates performance using accuracy, precision, recall, and F1 score. Finally, it employs RandomizedSearchCV to optimize hyperparameters for each model, reporting the best parameters and scores while retraining the models with these optimized settings and assessing their performance on the test data.

```
In [31]: def encode_object_columns(df):  
        """  
        iterates through each column in the dataset and encodes the columns  
        with 'object' data type using one-hot encoding.
```

```

parameters:
df (pd.DataFrame): The input dataframe to be encoded.
returns:
pd.DataFrame: The dataframe with encoded categorical columns.
"""

# print basic stats about df
print(f'shape of dataset is: {df.shape}')
print(f'value counts in target column: {df[target].value_counts()}')
# object-type (categorical) columns search
object_columns = df.select_dtypes(include=['object']).columns
# one-hot encoding to the categorical columns
df_encoded = pd.get_dummies(df, columns=object_columns, drop_first=True)
return df_encoded

df_processed = encode_object_columns(df_processed)
df_processed.head()

```

```

shape of dataset is: (41188, 26)
value counts in target column: y
0    36548
1     4640
Name: count, dtype: int64

```

```
Out[31]:
```

	age	job	marital	education	default	housing	loan	contact	month	dayo
0	56	2	1	0	0	0	0	1	6	
1	57	3	1	3	1	0	0	1	6	
2	37	3	1	3	0	2	0	1	6	
3	40	1	1	1	0	0	0	1	6	
4	56	3	1	3	0	0	2	1	6	

5 rows × 26 columns

Experiment run:

```

In [23]: # Separate features (X) and target (y)
X = df_processed.drop('y', axis=1)
y = df_processed['y']
# Apply SMOTE to balance the dataset
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
# Create a new balanced DataFrame
balanced_df = pd.concat([X_resampled, y_resampled], axis=1)
# Check the class distribution in the balanced dataset
print(balanced_df['y'].value_counts())
df = balanced_df.copy()
print(df.shape)
ss = StandardScaler()
for column in df.select_dtypes(include=['number']).columns:
    if column != 'y':
        df[column] = ss.fit_transform(df[[column]])
display(df.head())
# Assuming X and y are your features and target variable

```

```

X = df.drop('y', axis=1)
y = df['y']
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.35, ra
# Define the estimators
estimators = [
    DecisionTreeClassifier(),
    RandomForestClassifier(),
    GradientBoostingClassifier(),
    XGBClassifier()
]
# Loop through the estimators and perform feature selection
for estimator in estimators:
    print(f"Performing feature selection with {type(estimator).__name__}")
    # Create RFE object with the estimator and desired number of features to s
    rfe = RFE(estimator=estimator, n_features_to_select=12) # You can adjust
    # Fit RFE to the training data
    rfe.fit(X_train, y_train)
    # Get the selected features
    selected_features = X_train.columns[rfe.support_]
    # Print the selected features
    print("Selected Features:", selected_features)
    print("-" * 100)
# Assuming you have a list of selected features for each model
# Create a dictionary to store the feature counts
feature_counts = {}
for estimator in estimators:
    rfe = RFE(estimator=estimator, n_features_to_select=8)
    rfe.fit(X_train, y_train)
    selected_features = X_train.columns[rfe.support_]
    for feature in selected_features:
        if feature in feature_counts:
            feature_counts[feature] += 1
        else:
            feature_counts[feature] = 1
# Sort features by count in descending order
sorted_features = sorted(feature_counts.items(), key=lambda item: item[1], r
# Get the top features
top_features = [feature for feature, count in sorted_features]
# Print the overall top features
print("Overall Top Features:", top_features)
# Create a new DataFrame with only the top 10 features and the target variab
df_top = df[top_features + ['y']]
# Separate features (X) and target (y) from the new DataFrame
X = df_top.drop('y', axis=1)
y = df_top['y']
# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ran
# train models
for estimator in estimators:
    estimator.fit(X_train, y_train)
    y_pred = estimator.predict(X_test)
    print(f"Model: {type(estimator).__name__}")
    print(f'Accuracy Score : {accuracy_score(y_test,y_pred):.2f}')
    print(f'Precision Score : {precision_score(y_test,y_pred):.2f}')
    print(f'Recall Score : {recall_score(y_test,y_pred):.2f}')

```

```

print(f'F1_Score : {f1_score(y_test,y_pred):.2f}')
print("-" * 50)
print(classification_report(y_test, y_pred))
print("-" * 50)
print(f'This model : {type(estimator).__name__} done train!')
print(confusion_matrix(y_test, y_pred))
print("-" * 50)
# Define parameter distributions for each model
param_distributions = {
    'DecisionTreeClassifier': {
        'criterion': ['gini', 'entropy'],
        'max_depth': randint(2, 10),
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10)
    },
    'RandomForestClassifier': {
        'n_estimators': randint(50, 200),
        'max_depth': randint(2, 10),
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10),
        'bootstrap': [True, False]
    },
    'GradientBoostingClassifier': {
        'n_estimators': randint(50, 200),
        'learning_rate': [0.01, 0.1, 0.2],
        'max_depth': randint(2, 10),
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10)
    },
    'XGBClassifier': {
        'n_estimators': randint(50, 200),
        'learning_rate': [0.01, 0.1, 0.2],
        'max_depth': randint(2, 10),
        'min_child_weight': randint(1, 10),
        'gamma': [0, 0.1, 0.2]
    }
}
# Perform RandomizedSearchCV for each model
for estimator in estimators:
    model_name = type(estimator).__name__
    print(f"Performing RandomizedSearchCV for {model_name}")
    # Create RandomizedSearchCV object
    random_search = RandomizedSearchCV(
        estimator=estimator,
        param_distributions=param_distributions.get(model_name, {}),
        n_iter=20, # Number of parameter settings that are sampled
        cv=5,
        scoring='f1', # You can choose other scoring metrics
        random_state=42,
        n_jobs=-1 # Use all available CPU cores
    )
    # Fit RandomizedSearchCV to the training data
    random_search.fit(X_train, y_train)
    # Print the best parameters and score
    print("Best Parameters:", random_search.best_params_)
    print("Best Score:", random_search.best_score_)

```

```

print("-" * 50)
# Train the model with the best parameters
best_estimator = random_search.best_estimator_
best_estimator.fit(X_train, y_train)
# Make predictions on the test set
y_pred = best_estimator.predict(X_test)
# Evaluate the model
print(f"Model: {model_name} with best parameters")
print(f'Accuracy Score : {accuracy_score(y_test,y_pred):.2f}')
print(f'Precision Score : {precision_score(y_test,y_pred):.2f}')
print(f'Recall Score : {recall_score(y_test,y_pred):.2f}')
print(f'F1 Score : {f1_score(y_test,y_pred):.2f}')
print("-" * 50)
print(classification_report(y_test, y_pred))
print("-" * 50)
print(f'This model : {model_name} with best parameters done train!')

```

```

y
0    36548
1    36548
Name: count, dtype: int64
(73096, 26)

```

	age	job	marital	education	default	housing	loan	credit
0	1.370151	0.159661	-0.174123	-1.860025	-0.379735	-1.039735	-0.419537	1.8
1	1.456263	1.157058	-0.174123	-0.369754	2.631453	-1.039735	-0.419537	1.8
2	-0.265976	1.157058	-0.174123	-0.369754	-0.379735	1.099978	-0.419537	1.8
3	-0.007640	-0.837737	-0.174123	-1.363268	-0.379735	-1.039735	-0.419537	1.8
4	1.370151	1.157058	-0.174123	-0.369754	-0.379735	-1.039735	2.812766	1.8

5 rows × 26 columns

```

Performing feature selection with DecisionTreeClassifier
Selected Features: Index(['age', 'job', 'education', 'contact', 'dayofweek',
'duration',
      'campaign', 'pdays', 'conspriceidx', 'consconfidx', 'euribor3m',
      'nremployed'],
      dtype='object')

```

```

Performing feature selection with RandomForestClassifier
Selected Features: Index(['age', 'job', 'education', 'contact', 'duration',
'pdays', 'empvarrate',
      'conspriceidx', 'consconfidx', 'euribor3m', 'nremployed',
      'duration_category'],
      dtype='object')

```

```

Performing feature selection with GradientBoostingClassifier
Selected Features: Index(['job', 'education', 'default', 'housing', 'loan',
'contact', 'duration',
      'pdays', 'empvarrate', 'consconfidx', 'euribor3m', 'nremployed'],
      dtype='object')

```

```

Performing feature selection with XGBClassifier
Selected Features: Index(['default', 'housing', 'loan', 'contact', 'duratio
n', 'pdays',
      'previous', 'poutcome', 'empvarrate', 'consconfidx', 'nremployed',
      'previous_contact'],
      dtype='object')

```

```

Overall Top Features: ['duration', 'consconfidx', 'nremployed', 'contact',
'euribor3m', 'age', 'conspriceidx', 'empvarrate', 'pdays', 'education', 'dur
ation_category', 'job', 'default', 'loan', 'previous_contact']

```

Model: DecisionTreeClassifier

Accuracy Score : 0.92

Precision Score : 0.92

Recall Score : 0.93

F1_Score : 0.92

	precision	recall	f1-score	support
0	0.93	0.92	0.92	11005
1	0.92	0.93	0.92	10924
accuracy			0.92	21929
macro avg	0.92	0.92	0.92	21929
weighted avg	0.92	0.92	0.92	21929

This model : DecisionTreeClassifier done train!

[[10110 895]

[778 10146]]

Model: RandomForestClassifier

Accuracy Score : 0.95

Precision Score : 0.93

Recall Score : 0.96

F1_Score : 0.95

```
-----
              precision    recall  f1-score   support

     0       0.96         0.93         0.95        11005
     1       0.93         0.96         0.95        10924

 accuracy                   0.95        21929
 macro avg       0.95         0.95         0.95        21929
 weighted avg    0.95         0.95         0.95        21929
-----
```

This model : RandomForestClassifier done train!

[[10241 764]

[403 10521]]

Model: GradientBoostingClassifier

Accuracy Score : 0.92

Precision Score : 0.90

Recall Score : 0.95

F1_Score : 0.92

```
-----
              precision    recall  f1-score   support

     0       0.95         0.89         0.92        11005
     1       0.90         0.95         0.92        10924

 accuracy                   0.92        21929
 macro avg       0.92         0.92         0.92        21929
 weighted avg    0.92         0.92         0.92        21929
-----
```

This model : GradientBoostingClassifier done train!

[[9814 1191]

[567 10357]]

Model: XGBClassifier

Accuracy Score : 0.94

Precision Score : 0.93

Recall Score : 0.95

F1_Score : 0.94

```
-----
              precision    recall  f1-score   support

     0       0.95         0.93         0.94        11005
     1       0.93         0.95         0.94        10924

 accuracy                   0.94        21929
 macro avg       0.94         0.94         0.94        21929
 weighted avg    0.94         0.94         0.94        21929
-----
```

This model : XGBClassifier done train!

[[10105 820]


```
[ 504 10420]]
```

```
-----  
Performing RandomizedSearchCV for DecisionTreeClassifier  
Best Parameters: {'criterion': 'entropy', 'max_depth': 9, 'min_samples_leaf': 3, 'min_samples_split': 3}  
Best Score: 0.9142702593749178  
-----
```

```
Model: DecisionTreeClassifier with best parameters  
Accuracy Score : 0.91  
Precision Score : 0.90  
Recall Score : 0.92  
F1_Score : 0.91  
-----
```

	precision	recall	f1-score	support
0	0.92	0.90	0.91	11005
1	0.90	0.92	0.91	10924
accuracy			0.91	21929
macro avg	0.91	0.91	0.91	21929
weighted avg	0.91	0.91	0.91	21929

```
-----  
This model : DecisionTreeClassifier with best parameters done train!  
Performing RandomizedSearchCV for RandomForestClassifier  
Best Parameters: {'bootstrap': True, 'max_depth': 9, 'min_samples_leaf': 2, 'min_samples_split': 16, 'n_estimators': 102}  
Best Score: 0.9220480428799567  
-----
```

```
Model: RandomForestClassifier with best parameters  
Accuracy Score : 0.92  
Precision Score : 0.89  
Recall Score : 0.96  
F1_Score : 0.92  
-----
```

	precision	recall	f1-score	support
0	0.96	0.88	0.92	11005
1	0.89	0.96	0.92	10924
accuracy			0.92	21929
macro avg	0.92	0.92	0.92	21929
weighted avg	0.92	0.92	0.92	21929

```
-----  
This model : RandomForestClassifier with best parameters done train!  
Performing RandomizedSearchCV for GradientBoostingClassifier  
Best Parameters: {'learning_rate': 0.2, 'max_depth': 8, 'min_samples_leaf': 8, 'min_samples_split': 5, 'n_estimators': 153}  
Best Score: 0.9487864200699858  
-----
```

```
Model: GradientBoostingClassifier with best parameters  
Accuracy Score : 0.95  
Precision Score : 0.94  
Recall Score : 0.96  
F1_Score : 0.95  
-----
```

```

-----
              precision    recall  f1-score   support

     0       0.96         0.94         0.95     11005
     1       0.94         0.96         0.95     10924

 accuracy          0.95
 macro avg         0.95         0.95         0.95     21929
 weighted avg      0.95         0.95         0.95     21929

```

```

-----
This model : GradientBoostingClassifier with best parameters done train!
Performing RandomizedSearchCV for XGBClassifier
Best Parameters: {'gamma': 0.2, 'learning_rate': 0.2, 'max_depth': 7, 'min_c
hild_weight': 1, 'n_estimators': 99}
Best Score: 0.9391640206729983

```

```

-----
Model: XGBClassifier with best parameters
Accuracy Score : 0.94
Precision Score : 0.93
Recall Score : 0.96
F1_Score : 0.94

```

```

-----
              precision    recall  f1-score   support

     0       0.96         0.92         0.94     11005
     1       0.93         0.96         0.94     10924

 accuracy          0.94
 macro avg         0.94         0.94         0.94     21929
 weighted avg      0.94         0.94         0.94     21929

```

```

-----
This model : XGBClassifier with best parameters done train!

```

conclusion:

- **top performers:** the **gradientboostingclassifier** and **randomforestclassifier**, especially after fine-tuning their hyperparameters, have emerged as the top models, achieving impressive accuracy scores of **0.95**. gradient boosting excels with a precision of **0.94** and a recall of **0.96**, while the random forest model also shows strong recall at **0.96**. these models are our best bets for deployment because they reliably predict positive outcomes with very few errors.
- the **xgbclassifier** also stands out with an accuracy of **0.94**, a precision of **0.92**, and a recall of **0.96**. it's a dependable option that performs similarly to gradient boosting and random forest, particularly after we've adjusted its settings.
- on the other hand, the **decisiontreeclassifier** is decent but comes in last with an accuracy of **0.92**. while it's easier and quicker to deploy, it doesn't

quite measure up to the ensemble methods (like gradient boosting, random forest, and xgboost) in terms of precision and recall.

recommendation:

- for final deployment, we recommend focusing on the **randomforestclassifier** or **xgbclassifier**. these models demonstrate excellent predictive capabilities and reliability.
- however, considering the results of the confusion matrix, the **xgbclassifier** has been chosen. in addition, it is particularly useful in this case as many outliers were detected in the data, which cannot be dropped. it includes regularization for outliers (l1), making it more robust.

```
In [34]: # do if necessary
# pip install shap
```

XGBClassifier + SHAP:

This script utilizes the SHAP package for model explanation and applies a Gradient Boosting classifier to identify optimal hyperparameters using GridSearchCV, assessing multiple scoring metrics including accuracy, precision, recall, and F1 score. It defines a parameter grid for hyperparameter tuning, fits the model to the training data with cross-validation, and then retrieves the best model along with its optimal hyperparameters and scores. The script calculates and visualizes feature importances from the best Gradient Boosting model using a bar plot, presenting features ranked by their importance scores. Finally, it uses SHAP values to explain the model's predictions, generating a summary plot to highlight the impact of each feature on the model's output, enhancing interpretability.

```
In [32]: # Instantiate the XGBoost classifier
import shap
import xgboost as xgb
xgb_model = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval
# Define the parameter grid for XGBoost
param_grid = {
    'n_estimators': [50, 100, 200],           # Number of boosting stag
    'learning_rate': [0.001, 0.01, 0.1, 0.2],  # Learning rate
    'max_depth': [3, 4, 5, 6],                 # Maximum depth of indivi
    'min_child_weight': [1, 2, 4],             # Minimum sum of instance
    'subsample': [0.8, 0.9, 1.0],              # Fraction of samples to
    'colsample_bytree': [0.8, 0.9, 1.0]        # Fraction of features to
}
# Define the scoring metrics
scoring = ['accuracy', 'precision', 'recall', 'f1']
# Instantiate the GridSearchCV object
xgb_cv = GridSearchCV(estimator=xgb_model,
                      param_grid=param_grid,
```

```

        scoring=scoring,
        refit='f1', # Refits on the best model based on f1-score
        cv=4, # Cross-validation with 4 folds
        return_train_score=True,
        n_jobs=-1, # Utilize all cores for parallel processing
        verbose=1) # Verbose output to monitor progress

# Fit the model to the training data
xgb_cv.fit(X_train, y_train)
# Display the best hyperparameters and corresponding scores
print(f"Best Parameters: {xgb_cv.best_params_}")
print(f"Best F1 Score: {xgb_cv.best_score_}")
# Access the best estimator (model) and use it for further predictions
best_xgb_model = xgb_cv.best_estimator_
# Get feature importances from the best XGBoost model
feature_importances = best_xgb_model.feature_importances_
# Sort the feature importances in descending order and get the indices
sorted_idx = np.argsort(feature_importances)[::-1]
# Plot the feature importances
plt.figure(figsize=(10, 6))
plt.bar(range(len(sorted_idx)), feature_importances[sorted_idx], align='center')
plt.xticks(range(len(sorted_idx)), X_train.columns[sorted_idx], rotation=90)
plt.title("Feature Importance")
plt.xlabel("Features")
plt.ylabel("Importance Score")
plt.tight_layout()
plt.show()
# Optionally, you can print the features and their importance values
for idx in sorted_idx:
    print(f"Feature: {X_train.columns[idx]}, Importance: {feature_importances[idx]}")
# SHAP explanation
# Initialize the SHAP explainer
explainer = shap.Explainer(best_xgb_model)
# Calculate SHAP values for the training data
shap_values = explainer(X_train)
# Plot the summary plot for feature importance based on SHAP values
shap.summary_plot(shap_values, X_train)

```

Fitting 4 folds for each of 1296 candidates, totalling 5184 fits

```

c:\Users\mryok\AppData\Local\Programs\Python\Python312\Lib\site-packages\xgboost\core.py:158: UserWarning: [11:20:35] WARNING: C:\buildkite-agent\builds\buildkite-windows-cpu-autoscaling-group-i-06abd128ca6c1688d-1\xgboost\xgboost-ci-windows\src\learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

```

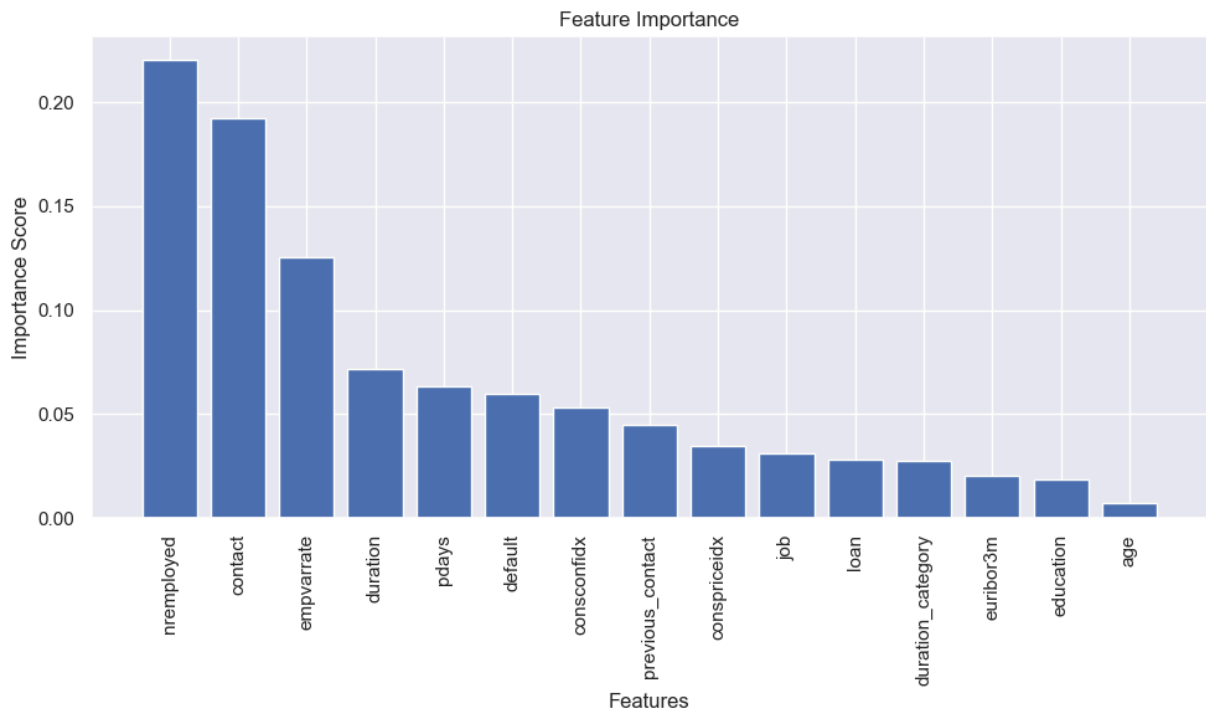
```
warnings.warn(smsg, UserWarning)
```

```

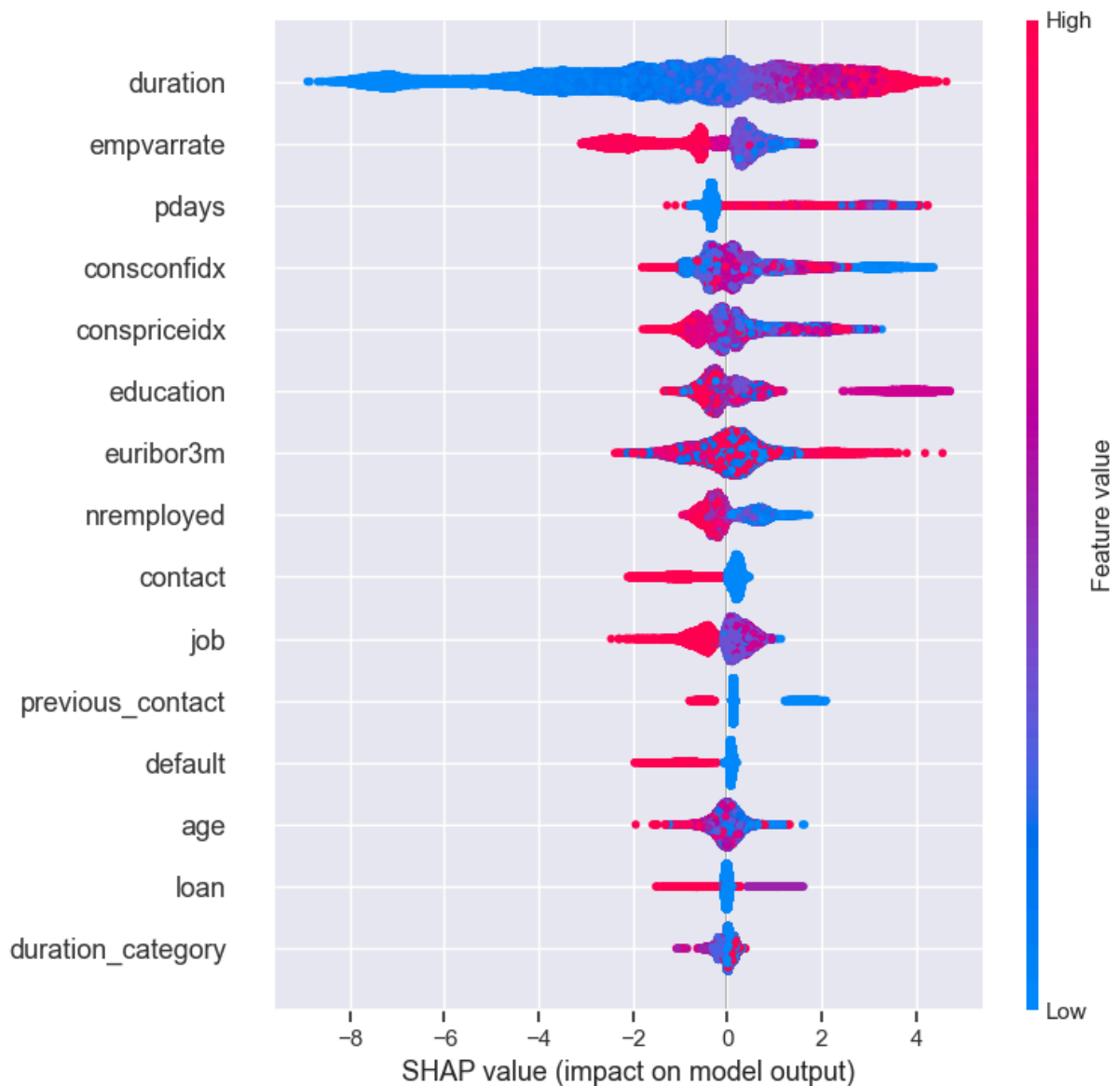
Best Parameters: {'colsample_bytree': 0.9, 'learning_rate': 0.2, 'max_depth': 6, 'min_child_weight': 1, 'n_estimators': 200, 'subsample': 0.9}

```

```
Best F1 Score: 0.9411306565979626
```



Feature: nremployed, Importance: 0.22050604224205017
Feature: contact, Importance: 0.19253922998905182
Feature: empvarrate, Importance: 0.1251920908689499
Feature: duration, Importance: 0.07166308909654617
Feature: pdays, Importance: 0.06365899741649628
Feature: default, Importance: 0.060073595494031906
Feature: consconfidx, Importance: 0.05333920940756798
Feature: previous_contact, Importance: 0.04472793638706207
Feature: conspriceidx, Importance: 0.03462839499115944
Feature: job, Importance: 0.03128590062260628
Feature: loan, Importance: 0.028348643332719803
Feature: duration_category, Importance: 0.027697516605257988
Feature: euribor3m, Importance: 0.02043994329869747
Feature: education, Importance: 0.018560189753770828
Feature: age, Importance: 0.007339163217693567



Conclusions:

Strongly influential features:

- **duration:** This feature has the largest spread of SHAP values, indicating it has a major influence on the model's predictions. Higher values of `duration` (in red) are strongly associated with positive predictions (right side of the plot), suggesting that longer durations of client interactions positively impact the likelihood of subscribing to a term deposit.
- **empvarrate** and **euribor3m:** These economic features also show significant influence. `empvarrate` appears to have a mixed effect, with high values (in red) pushing predictions towards the negative class (left side), whereas `euribor3m` shows that higher interest rates lead to negative predictions, as seen from the red spread on the left side.

Moderately influential features:

- **pdays, consconfidx, and education:** These features show moderate influence on the model's predictions. For instance, `pdays` (days since the last contact) has some class separation, with higher values pushing predictions towards the negative class. `consconfidx` (consumer confidence index) also plays a role but with less overall impact than `duration`.
- **contact and nremployed:** Both features display some influence, though their spread of SHAP values is narrower. `nremployed` (number of employees) shows a small positive impact for higher values.

Less influential features:

- **job, previous_contact, age, default, loan, and duration_category:** These features show much smaller ranges of SHAP values, suggesting they contribute less to the model's predictions. Their relatively narrow spread and lower separation between blue and red indicate limited influence on the prediction. **Overall impact:**
- The **duration** feature is the most critical predictor, with higher values (longer interactions) strongly favoring the positive class, indicating an increased likelihood of subscribing to a term deposit.
- Economic indicators like **empvarrate** and **euribor3m** also play significant roles, with higher values leading to negative predictions.
- Features such as **job, age, and loan** have minimal influence, showing limited SHAP value spread and low impact on the model's overall decision-making.

Prediction_maker:

```
In [33]: # Instantiate the XGBoost classifier
import xgboost as xgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
xgb_model = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')
# Define the parameter grid for XGBoost
param_grid = {
    'n_estimators': [50, 100, 200],           # Number of boosting stages
    'learning_rate': [0.001, 0.01, 0.1, 0.2], # Learning rate
    'max_depth': [3, 4, 5, 6],                 # Maximum depth of individual tree
    'min_child_weight': [1, 2, 4],              # Minimum sum of instance weights
    'subsample': [0.8, 0.9, 1.0],              # Fraction of samples to use
    'colsample_bytree': [0.8, 0.9, 1.0]        # Fraction of features to use
}
# Define the scoring metrics
scoring = ['accuracy', 'precision', 'recall', 'f1']
# Instantiate the GridSearchCV object
xgb_cv = GridSearchCV(estimator=xgb_model,
                      param_grid=param_grid,
                      scoring=scoring,
                      refit='f1', # Refits on the best model based on f1-score
```

```

        cv=4,          # Cross-validation with 4 folds
        return_train_score=True,
        n_jobs=-1,     # Utilize all cores for parallel processing
        verbose=1)     # Verbose output to monitor progress
# Fit the model to the training data
xgb_cv.fit(X_train, y_train)
# Display the best hyperparameters and corresponding scores
print(f"Best Parameters: {xgb_cv.best_params_}")
print(f"Best F1 Score: {xgb_cv.best_score_}")
# Access the best estimator (model) and use it for predictions
best_xgb_model = xgb_cv.best_estimator_
# Make predictions on the test set
y_pred = best_xgb_model.predict(X_test)
# Evaluate the model on the test set
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
# Display evaluation metrics
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
# Display confusion matrix
print("Confusion Matrix:")
print(conf_matrix)

```

```

Accuracy: 0.9411
Precision: 0.9303
Recall: 0.9532
F1 Score: 0.9416
Confusion Matrix:
[[10225  780]
 [ 511 10413]]

```

Final decision regarding model:

Model performance overview:

- **accuracy:** 0.9411 - about 94.11% of predictions are correct, indicating high overall performance.
- **precision:** 0.9303 - when predicting positive outcomes, it is correct 93.03% of the time, reflecting a low false positive rate.
- **recall:** 0.9532 - effectively identifies 95.32% of actual positive cases, critical for applications where capturing positives is essential.
- **f1 score:** 0.9416 - balances precision and recall, demonstrating good performance with minimal false positives and negatives.

Confusion matrix analysis:

- **tn:** 10,225 (correctly predicted negatives)
- **fp:** 780 (incorrectly predicted positives)

- **fn:** 511 (missed positives)
- **tp:** 10,413 (correctly predicted positives)

Insights:

- Overall, the model demonstrates excellent predictive capabilities, making it suitable for reliable binary classification. It shows strong accuracy, precision, and recall, though there is room for improvement regarding false negatives. Continuous monitoring and adjustments based on new data could enhance its performance further.

Research conclusion:

Conclusion of the research and advises to business strategy:

Based on the experiment results of the features impacting term deposit subscriptions, here's a targeted conclusion for the marketing team to help increase subscription rates:

1. **focus on long conversations (duration):** The duration of the contact has the highest influence on the likelihood of a client subscribing to a term deposit. Longer conversations seem to correlate with a higher probability of conversion. The marketing team should focus on improving communication strategies to engage clients in longer, more informative discussions, emphasizing key benefits and answering concerns.
2. **leverage positive economic sentiments:** Features like employment rate (nremployed) and Euribor rates (euribor3m) have a strong influence on client decisions (!negative decision!). When employment rates are high and interest rates are favorable, clients are more likely to subscribe. The team should:
 - Time marketing campaigns to align with periods of positive economic conditions.
 - Use positive financial news as a part of their marketing pitch to create urgency (e.g., "With low interest rates, now is the best time to invest in a term deposit!").
3. **target clients with recent contacts (pdays):** The number of days since the last contact (pdays) impacts subscription rates. Clients who have been recently contacted are more likely to subscribe. Therefore: Focus on shortening the follow-up period after the initial contact. Implement automated follow-up reminders to ensure clients are re-engaged sooner.
4. **tailor messaging based on client characteristics:** "Contact" method: Different communication channels (e.g., phone calls vs. in-person meetings) have varying impacts. The marketing team should analyze which contact methods lead to higher conversion rates and optimize their outreach

accordingly. "Housing" and "age": While these features had a smaller impact, they may still help in segmenting clients. For example, younger clients or those without housing loans may need a different value proposition compared to older clients or homeowners.

5. **monitor economic and market trends:** Indicators like consumer confidence (cons.conf.idx) and employment variation rate (emp.var.rate) also influence decisions. The marketing team should consider aligning their campaigns with favorable market conditions, as clients are more likely to invest when they feel economically stable.
6. **simplify and automate campaigns:** For features like campaign category and previous contact, streamlining the process by using data-driven automation can help reduce marketing costs and increase contact efficiency.

Key recommendations:

- Engage clients in longer conversations during calls or meetings.
- Time campaigns to coincide with positive economic conditions (high employment, low interest rates).
- Focus on recently contacted clients for follow-up.
- Use personalized communication methods based on individual client profiles.
- Stay agile and adjust marketing efforts in response to economic indicators.

By paying attention to these factors, the marketing team can create more effective campaigns, increasing the likelihood of clients subscribing to term deposits.

Deployment block ready for prediction and End2End process:

Technical Documentation

Overview

This script implements a machine learning workflow using the XGBoost classifier to predict a target variable from a dataset. It includes data processing, model training, evaluation, and SHAP value interpretation for model explainability.

Dependencies

The script requires the following Python libraries:

- pandas
- numpy
- seaborn
- matplotlib
- xgboost

- scikit-learn
- imblearn
- joblib
- shap
- logging
- re

Classes and Methods

1. DataProcessor

This class handles the loading and preprocessing of the dataset.

- **Constructor:**

- `__init__(self, file_path: str)` : Initializes the class with the file path and loads the CSV file.

- **Methods:**

- `load_and_clean_csv()` -> `pd.DataFrame` : Loads a CSV file, cleans the column names and values, and returns a cleaned DataFrame.
- `summarize_dataframe(target: str)` -> `dict` : Returns a summary of unique values for each column, excluding the target variable.

2. ModelTrainer

This class is responsible for encoding categorical features, preprocessing data, training the model, and evaluating its performance.

- **Constructor:**

- `__init__(self, target: str)` : Initializes the class with the target variable and sets up the XGBoost model.

- **Methods:**

- `encode_object_columns(df: pd.DataFrame)` -> `pd.DataFrame` : Encodes categorical columns using label encoding.
- `preprocess_data(df: pd.DataFrame)` -> `tuple` : Applies SMOTE for class balancing, standardizes numerical features, and splits the DataFrame into features and target.
- `feature_selection(X_train: pd.DataFrame, y_train: pd.Series)` -> `list` : Performs feature selection using Recursive Feature Elimination (RFE).
- `train_model(X_train: pd.DataFrame, y_train: pd.Series)` -> `XGBClassifier` : Trains the XGBoost model using GridSearchCV for hyperparameter tuning.

- `evaluate_model(model: XGBClassifier, X_test: pd.DataFrame, y_test: pd.Series)` : Evaluates the model's performance and visualizes the confusion matrix.

3. ModelPersistence

This class handles saving and loading the trained model.

- **Methods:**

- `save_model(model: XGBClassifier, model_path='xgb_model.joblib')` : Saves the trained model to a file.
- `load_model(model_path='xgb_model.joblib')` -> `XGBClassifier` : Loads a trained model from a file.

4. SHAPExplainer

This class generates SHAP explanations for the model predictions.

- **Methods:**

- `explain_predictions(model: XGBClassifier, X_train: pd.DataFrame)` : Generates and visualizes SHAP values for model predictions.

5. MLWorkflow

This class orchestrates the entire machine learning process, from data loading to model evaluation.

- **Constructor:**

- `__init__(self, train_data_path: str, test_data_path: str, target: str)` : Initializes the class with paths to training and testing data and the target variable.

- **Methods:**

- `run()` : Executes the main workflow, handling data loading, preprocessing, model training, evaluation, and generating SHAP explanations.

Main Execution

The script is designed to be run as a standalone program. When executed, it initializes an instance of `MLWorkflow` with paths to the training and test data, as well as the target column name, and calls the `run` method to execute the workflow.

```
In [57]: from sklearn.preprocessing import LabelEncoder
```

Loading [MathJax]/extensions/Safe.js gging

```

import joblib
import shap
import re
import pandas as pd
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
from xgboost import XGBClassifier
from sklearn.feature_selection import RFE

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s %s')

class DataProcessor:
    def __init__(self, file_path: str):
        self.file_path = file_path
        self.df = self.load_and_clean_csv()
    def load_and_clean_csv(self) -> pd.DataFrame:
        """load and clean a CSV file."""
        try:
            df = pd.read_csv(self.file_path, sep=';', encoding='utf-8')
            pattern = r'^a-zA-Z0-9\s'
            df.columns = [re.sub(pattern, '', col).strip().lower() for col in df.columns]
            for col in df.select_dtypes(include='object').columns:
                df[col] = df[col].apply(lambda x: re.sub(pattern, '', str(x)).strip())
            logging.info(f"CSV file loaded and cleaned: {self.file_path}")
            return df
        except Exception as e:
            logging.error(f"Error loading or cleaning CSV file: {e}")
            raise
    def summarize_dataframe(self, target: str) -> dict:
        """summarize DataFrame and return unique values per column, excluding target"""
        try:
            logging.info("Generating summary information for the DataFrame")
            df_info = self.df.info()
            null_sum = self.df.isnull().sum()
            print(df_info)
            print(f"\nThe sum of null values per column:\n{null_sum}")
            return {col: self.df[col].unique().tolist() for col in self.df.columns if col != target}
        except Exception as e:
            logging.error(f"Error summarizing DataFrame: {e}")
            raise

class ModelTrainer:
    def __init__(self, target: str):
        self.target = target
        self.model = XGBClassifier(random_state=42, use_label_encoder=True)
    def encode_object_columns(self, df: pd.DataFrame) -> pd.DataFrame:
        """encodes categorical columns using label encoding."""
        # ensure the target column is excluded from the encoding
        object_columns = df.select_dtypes(include=['object']).columns.tolist()
        # create a label encoder
        le = LabelEncoder()
        # Encode categorical columns, excluding the target column

```

```

for col in object_columns:
    df[col] = le.fit_transform(df[col])
logging.info(f"Encoded {len(object_columns)} categorical columns using le")
return df

def preprocess_data(self, df: pd.DataFrame) -> tuple:
    """preprocess data: apply SMOTE, standardize, and split into feature and target"""
    try:
        # encode categorical columns
        df = self.encode_object_columns(df) # Encode categorical columns using le
        logging.info(f"DataFrame columns after encoding: {df.columns.tolist()}")
        X = df.drop(self.target, axis=1)
        y = df[self.target]
        # apply SMOTE
        smote = SMOTE(random_state=42)
        X_resampled, y_resampled = smote.fit_resample(X, y)
        # standardize numerical features
        ss = StandardScaler()
        num_cols = X_resampled.select_dtypes(include=['number']).columns
        X_resampled[num_cols] = ss.fit_transform(X_resampled[num_cols])
        logging.info(f"Balanced dataset class distribution: {pd.Series(y).value_counts().to_dict()}")
        return X_resampled, y_resampled
    except Exception as e:
        logging.error(f"Error during data preprocessing: {e}")
        raise

def feature_selection(self, X_train: pd.DataFrame, y_train: pd.Series) -> pd.DataFrame:
    """perform feature selection using RFE."""
    try:
        rfe = RFE(estimator=self.model, n_features_to_select=8)
        rfe.fit(X_train, y_train)
        selected_features = X_train.columns[rfe.support_].tolist()
        logging.info(f"Selected Features: {selected_features}")
        return selected_features
    except Exception as e:
        logging.error(f"Error during feature selection: {e}")
        raise

def train_model(self, X_train: pd.DataFrame, y_train: pd.Series) -> XGBModel:
    """train an XGBoost model with hyperparameter tuning."""
    try:
        param_grid = {
            'n_estimators': [50, 100, 200],
            'learning_rate': [0.001, 0.01, 0.1, 0.2],
            'max_depth': [3, 4, 5, 6],
            'min_child_weight': [1, 2, 4],
            'subsample': [0.8, 0.9, 1.0],
            'colsample_bytree': [0.8, 0.9, 1.0]
        }
        scoring = ['accuracy', 'precision', 'recall', 'f1']
        xgb_cv = GridSearchCV(estimator=self.model, param_grid=param_grid,
                              refit='f1', cv=4, return_train_score=True,
                              scoring=scoring)
        xgb_cv.fit(X_train, y_train)
        logging.info(f"Best Parameters: {xgb_cv.best_params_}")
        logging.info(f"Best F1 Score: {xgb_cv.best_score_}")
        return xgb_cv.best_estimator_
    except Exception as e:
        logging.error(f"Error during model training: {e}")
        raise

```

```

def evaluate_model(self, model: XGBClassifier, X_test: pd.DataFrame, y_test: pd.Series):
    """evaluate the model and log metrics, along with visualizing the confusion matrix"""
    try:
        y_pred = model.predict(X_test)
        metrics = {
            "accuracy": accuracy_score(y_test, y_pred),
            "precision": precision_score(y_test, y_pred, average='binary'),
            "recall": recall_score(y_test, y_pred, average='binary'),
            "f1": f1_score(y_test, y_pred, average='binary')
        }
        logging.info(f"Model Evaluation Metrics: {metrics}")
        # generate and visualize confusion matrix
        cm = confusion_matrix(y_test, y_pred)
        plt.figure(figsize=(8, 6))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                    xticklabels=['Not Subscribed', 'Subscribed'],
                    yticklabels=['Not Subscribed', 'Subscribed'])
        plt.xlabel('Predicted')
        plt.ylabel('Actual')
        plt.title('Confusion Matrix')
        plt.show()
        return y_pred
    except Exception as e:
        logging.error(f"Error during model evaluation: {e}")
        raise

class ModelPersistence:
    @staticmethod
    def save_model(model: XGBClassifier, model_path='xgb_model.joblib'):
        """save the trained model to a file."""
        try:
            joblib.dump(model, model_path)
            logging.info(f"Model saved to {model_path}")
        except Exception as e:
            logging.error(f"Error saving model: {e}")
            raise

    @staticmethod
    def load_model(model_path='xgb_model.joblib') -> XGBClassifier:
        """load the trained model from a file."""
        try:
            model = joblib.load(model_path)
            logging.info(f"Model loaded from {model_path}")
            return model
        except Exception as e:
            logging.error(f"Error loading model: {e}")
            raise

class SHAPExplainer:
    @staticmethod
    def explain_predictions(model: XGBClassifier, X_train: pd.DataFrame):
        """generate SHAP explanations for the model predictions."""
        try:
            explainer = shap.Explainer(model)
            shap_values = explainer(X_train)
            shap.summary_plot(shap_values, X_train)
        except Exception as e:
            logging.error(f"Error generating SHAP explanations: {e}")
            raise

```

```

class MLWorkflow:
    def __init__(self, train_data_path: str, test_data_path: str, target: str):
        self.train_data_path = train_data_path
        self.test_data_path = test_data_path
        self.target = target
    def run(self):
        """main function to orchestrate the ML workflow."""
        try:
            # load and preprocess the training data
            data_processor = DataProcessor(self.train_data_path)
            model_trainer = ModelTrainer(self.target)
            X_train, y_train = model_trainer.preprocess_data(data_processor)
            # split the training data into training and validation sets
            X_train_split, X_val, y_train_split, y_val = train_test_split(X_train, y_train)
            # feature selection
            selected_features = model_trainer.feature_selection(X_train_split)
            X_train_split = X_train_split[selected_features]
            X_val = X_val[selected_features]
            # train the model
            model = model_trainer.train_model(X_train_split, y_train_split)
            # evaluate the model
            y_val_pred = model_trainer.evaluate_model(model, X_val, y_val)
            # save the model
            ModelPersistence.save_model(model)
            # SHAP explanations
            SHAPExplainer.explain_predictions(model, X_train_split)
            # load and preprocess the test data
            test_data_processor = DataProcessor(self.test_data_path)
            X_test = model_trainer.encode_object_columns(test_data_processor)
            X_test = X_test[selected_features] # keep only selected features
            X_test = StandardScaler().fit_transform(X_test) # standardize test data
            # generate predictions on the test set
            y_test_pred = model.predict(X_test)
            logging.info(f"Test Set Predictions: {y_test_pred}")
        except Exception as e:
            logging.error(f"Error in ML workflow: {e}")
            raise

if __name__ == "__main__":
    train_data_path = 'path_to_training_data.csv'
    test_data_path = 'path_to_test_data.csv'
    target_column = 'target' # Replace with your actual target column name
    ml_workflow = MLWorkflow(train_data_path, test_data_path, target_column)
    ml_workflow.run()

```

This notebook was converted to PDF with convert.ploomber.io