

Predicting Diabetes

▼ Overview

▼ Intro - The Cost of Diabetes

According to the International Diabetes Federation, in 2021, an estimated 537 million people worldwide had diabetes, and this number is projected to rise to 642 million by 2040. This is roughly 1 in 15 people and around 7% of the total population living with Diabetes. In the United States, the American Diabetes Association reports that the total estimated cost of diagnosed diabetes was \$327 billion in 2017, including direct medical costs and reduced productivity.

Individuals with diabetes tend to have higher healthcare expenses compared to those without the condition. Diabetes also incurs indirect costs, such as lost productivity and disability. These costs arise from missed workdays, reduced productivity at work, early retirement, and disability due to diabetes-related complications.

▼ Business Problem

An insurance company wants to develop a predictive model to assess the risk of diabetes among their policyholders based on a limited set of available data points. By accurately identifying individuals at high risk of developing diabetes, the company aims to take proactive measures to reduce healthcare costs and improve the overall health outcomes of their customers.

The challenge for the company is to build a robust and accurate predictive model that can handle the complexity and non-linear relationships between the available data points and the risk of developing diabetes. The model will consider factors such as age, gender, BMI, hypertension status, heart disease history, smoking history, HbA1c level, and blood glucose level. We will use a classification model to predict diabetes within the population of interest.

My background and work history has been in healthcare which makes this an interesting problem for me. Being able to accurately predict risk within a population and provide resources and preventive measures are important now more than ever.

▼ Evaluation Metrics

To evaluate the different approaches in our classification, we will focus on 2 metrics, `Precision`, and `F1 score`. Using precision and F1 score as primary metrics in a diabetes classification task is important for several reasons:

Precision focuses on the accuracy of positive predictions, specifically the ratio of true positives to the sum of true positives and false positives. Maximizing precision helps ensure that the patients identified as positive for diabetes are highly likely to be true positives, reducing the risk of false positives.

F1 score is a balanced metric that considers both precision and recall. By optimizing for F1 score, we are aiming to achieve a balance between correctly identifying positive cases and minimizing false negatives.

In the context of diabetes classification, the consequences of false positives and false negatives can be significant. False positives may lead to unnecessary medical interventions or treatments for patients who are not actually diabetic, causing unnecessary costs and potential harm. False negatives, on the other hand, can result in undiagnosed diabetes cases going untreated, leading to potential health risks and complications. By focusing on precision and F1 score, you aim to strike the right balance between identifying true positive cases and minimizing false predictions, ultimately improving the overall effectiveness of the classification model for diabetes diagnosis.

▼ About the Data

The Diabetes prediction dataset is a collection of medical and demographic data from patients, along with their diabetes status (positive or negative). The data includes features such as age, gender, body mass index (BMI), hypertension, heart disease, smoking history, HbA1c level, and blood glucose level. This dataset can be used to build machine learning models to predict diabetes in patients based on their medical history and demographic information. This can be useful for healthcare professionals in identifying patients who may be at risk of developing

diabetes and in developing personalized treatment plans. Additionally, the dataset can be used by researchers to explore the relationships between various medical and demographic factors and the likelihood of developing diabetes.

Our dataset came from a Kaggle contributor located at this URL (<https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset>). We downloaded the data and added to our GitHub repo.

- `gender` - Gender refers to the biological sex of the individual, which can have an impact on their susceptibility to diabetes. There are three categories in it male, female and other.
- `age` - Age is an important factor as diabetes is more commonly diagnosed in older adults. Age ranges from 0-80 in our dataset.
- `hypertension` - Hypertension is a medical condition in which the blood pressure in the arteries is persistently elevated. It has values a 0 or 1 where 0 indicates they don't have hypertension and for 1 it means they have hypertension.
- `heart_disease` - Heart disease is another medical condition that is associated with an increased risk of developing diabetes. It has values a 0 or 1 where 0 indicates they don't have heart disease and for 1 it means they have heart disease.
- `smoking_history` - Smoking history is also considered a risk factor for diabetes and can exacerbate the complications associated with diabetes. In our dataset we have 5 categories i.e not current, former, No Info, current, never and ever.
- `bmi` - BMI (Body Mass Index) is a measure of body fat based on weight and height. Higher BMI values are linked to a higher risk of diabetes. The range of BMI in the dataset is from 10.16 to 71.55. BMI less than 18.5 is underweight, 18.5-24.9 is normal, 25-29.9 is overweight, and 30 or more is obese.
- `HbA1c_level` - HbA1c (Hemoglobin A1c) level is a measure of a person's average blood sugar level over the past 2-3 months. Higher levels indicate a greater risk of developing diabetes. Mostly more than 6.5% of HbA1c Level indicates diabetes.
- `blood_glucose_level` - Blood glucose level refers to the amount of glucose in the bloodstream at a given time. High blood glucose levels are a key indicator of diabetes.
- `diabetes` - Diabetes is the target variable being predicted, with values of 1 indicating the presence of diabetes and 0 indicating the absence of diabetes.

▼ Data Acquisition

▼ Importing our packages for EDA

```

1 #importing necessary packages
2 import pandas as pd
3 import numpy as np
4 from matplotlib import pyplot as plt
5 import matplotlib.ticker as mtick
6 from matplotlib.colors import ListedColormap
7 import seaborn as sns
8 import statsmodels.api as sm
9 import requests
10 from zipfile import ZipFile
11 from io import BytesIO
12 from tabulate import tabulate
13 import pandas_gbq as gbq
14
15 %matplotlib inline
16
17 from sklearn.linear_model import LinearRegression, LogisticRegression, RidgeClassifier
18 from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV, StratifiedKFold
19 from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler, FunctionTransformer
20 from sklearn.pipeline import Pipeline
21 from sklearn.impute import MissingIndicator, SimpleImputer
22 from sklearn.compose import ColumnTransformer, make_column_transformer, make_column_selector
23 from sklearn.dummy import DummyClassifier
24 import sklearn.metrics
25 from sklearn.metrics import roc_auc_score
26 from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix, RocCurveDisplay, classification_report
27 from sklearn.metrics import precision_score, recall_score, accuracy_score, f1_score, precision_recall_curve, roc_curve, roc_auc_score
28 from sklearn.neighbors import KNeighborsClassifier, NearestNeighbors
29 from sklearn.ensemble import AdaBoostRegressor, GradientBoostingRegressor, AdaBoostClassifier, GradientBoostingClassifier, RandomForestClassifier
30 from sklearn.feature_selection import RFECV
31 import xgboost as xgb
32 from sklearn.neural_network import MLPClassifier
33 from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier, plot_tree
34 from imblearn.over_sampling import SMOTE
35 from imblearn.pipeline import make_pipeline, Pipeline as ImPipeline
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```



▼ Setting up our data file pull from GitHub

```

1 #get the URL of the zip file on GitHub.
2 url = 'https://github.com/heathlikethecandybar/phase_5/raw/main/data/diabetes-prediction-dataset.zip'
3
4 #send a GET request to download the zip file.
5 response = requests.get(url)
6
7 #read the zip file content.
8 zip_file = ZipFile(BytesIO(response.content))
9
10 #extract the CSV file from the zip file.
11 csv_file = zip_file.namelist()[0] # assumes only one file in the zip folder
12

```

▼ Importing data

```
1 #attaching our csv to a pandas dataframe
2 df = pd.read_csv(zip_file.open(csv_file))
```

▼ Exploratory Data Analysis

```
1 #quick look at our data
2 df.head()
```

	gender	age	hypertension	heart_disease	smoking_history	bmi	HbA1c_le
0	Female	80.0	0	1	never	25.19	
1	Female	54.0	0	0	No Info	27.32	
2	Male	28.0	0	0	never	27.32	
3	Female	36.0	0	0	current	23.45	
4	Male	76.0	1	1	current	20.14	

```
1 #loop through each column in the df
2 for column in df.columns:
3     distinct_values = df[column].unique()
4     print(f"Column: {column}")
5     print(f"Distinct Values: {distinct_values}")
6     print("-----")
```

```
Column: gender
Distinct Values: ['Female' 'Male' 'Other']
```

```
Column: age
Distinct Values: [80.  54.  28.  36.  76.  20.  44.  79.  42.  32.  53.  78.
 67.  15.  37.  40.  5.  69.  72.  4.  30.  45.  43.  50.
 41.  26.  34.  73.  77.  66.  29.  60.  38.  3.  57.  74.
 19.  46.  21.  59.  27.  13.  56.  2.  7.  11.  6.  55.
 9.  62.  47.  12.  68.  75.  22.  58.  18.  24.  17.  25.
 0.08 33.  16.  61.  31.  8.  49.  39.  65.  14.  70.  0.56
 48.  51.  71.  0.88 64.  63.  52.  0.16 10.  35.  23.  0.64
 1.16 1.64 0.72 1.88 1.32 0.8  1.24 1.  1.8  0.48 1.56 1.08
 0.24 1.4  0.4  0.32 1.72 1.48]
```

```
Column: hypertension
Distinct Values: [0 1]
```

```
Column: heart_disease
Distinct Values: [1 0]
```

```
Column: smoking_history
Distinct Values: ['never' 'No Info' 'current' 'former' 'ever' 'not current']
```

```
Column: bmi
Distinct Values: [25.19 27.32 23.45 ... 59.42 44.39 60.52]
```

```
Column: HbA1c_level
Distinct Values: [6.6 5.7 5.  4.8 6.5 6.1 6.  5.8 3.5 6.2 4.  4.5 9.  7.  8.8 8.2 7.5 6.8]
```

```
Column: blood_glucose_level
Distinct Values: [140 80 158 155 85 200 145 100 130 160 126 159 90 260 220 300 280 240]
```

```
Column: diabetes
Distinct Values: [0 1]
```

Looks like we will have 7 independent variables, and 1 dependent variable (target) which is our diabetes column. Hypertension, heart disease, and smoking history are categorical values. In which we may need to one encode.

```
1 #taking a look at our data types
2 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 9 columns):
#   Column                Non-Null Count  Dtype
---  -
0   gender                100000 non-null object
1   age                   100000 non-null float64
```

```

2 hypertension      100000 non-null int64
3 heart_disease      100000 non-null int64
4 smoking_history     100000 non-null object
5 bmi                100000 non-null float64
6 HbA1c_level         100000 non-null float64
7 blood_glucose_level 100000 non-null int64
8 diabetes            100000 non-null int64
dtypes: float64(3), int64(4), object(2)
memory usage: 6.9+ MB

```

```

1 #look for missing records
2 df.isna().sum()

```

```

gender      0
age          0
hypertension 0
heart_disease 0
smoking_history 0
bmi          0
HbA1c_level  0
blood_glucose_level 0
diabetes     0
dtype: int64

```

No missing records which is good. 100k records, which is a good sized data set. I wish there was more interesting columns, however, these are the core tenants for predicting diabetes.

```

1 #summary stats
2 df.describe()

```

	age	hypertension	heart_disease	bmi	HbA1c_level
count	100000.000000	100000.000000	100000.000000	100000.000000	100000.000000
mean	41.885856	0.07485	0.039420	27.320767	5.527507
std	22.516840	0.26315	0.194593	6.636783	1.070672
min	0.080000	0.00000	0.000000	10.010000	3.500000
25%	24.000000	0.00000	0.000000	23.630000	4.800000
50%	43.000000	0.00000	0.000000	27.320000	5.800000
75%	60.000000	0.00000	0.000000	29.580000	6.200000
max	80.000000	1.00000	1.000000	95.690000	9.000000

Roughly 8.5% of the population in this dataset has been diagnosed with Diabetes. It wasn't explicitly stated, but based on this number, I am assuming that this is based on type 2 diabetes diagnoses. Average age is 42 years of age, roughly 7.5% of the population has been diagnosed with hypertension, and 4% diagnosed with heart disease. The average Body Mass Index (BMI) is 27, which is considered overweight. The blood glucose level is 138 and HbA1c level is 5.5 on average. For non-diabetics, the normal HbA1c level ranges between 4% and 5.6%. Levels between 5.7% and 6.4% indicate prediabetes and a greater possibility of diabetes. HbA1c of 6.5% or greater indicates diabetes. Also looks like we have some weird decimals for our ages. I am going to round those down to the nearest whole number.

```

1 #rounding down the age to nearest whole number and adjusting data type to int
2 df.loc[:, 'age'] = np.floor(df.loc[:, 'age']).astype(int);

```

```

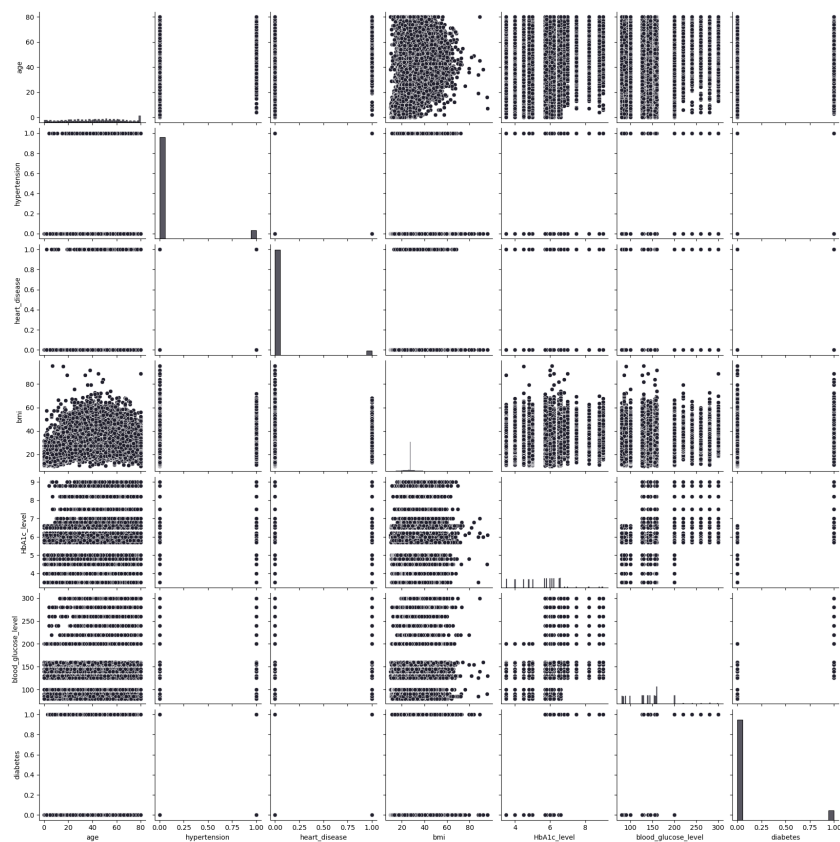
<ipython-input-372-aff4eeaa9027>:2: DeprecationWarning: In a future version, `df.iloc[:, i] = newvals` will attempt to set the
df.loc[:, 'age'] = np.floor(df.loc[:, 'age']).astype(int);

```

```

1 #create the pair plot to look at distributions
2 sns.pairplot(df);

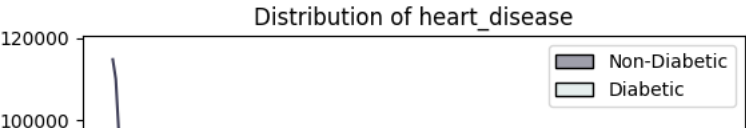
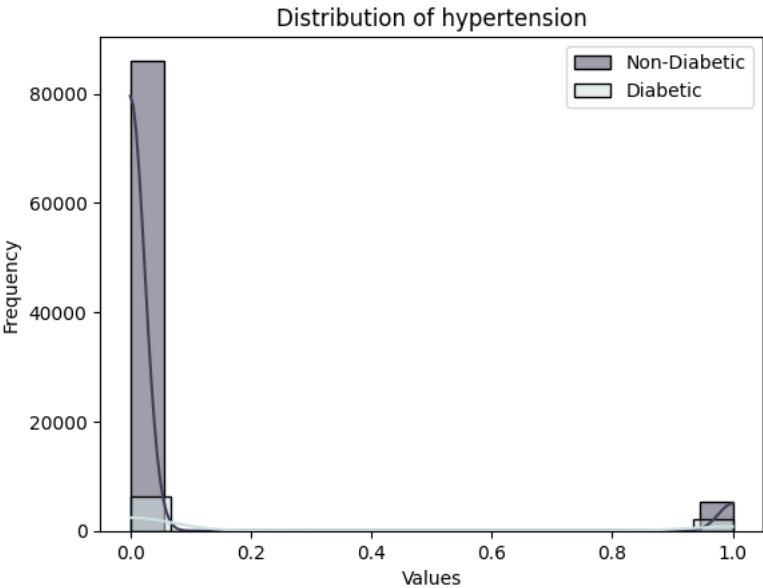
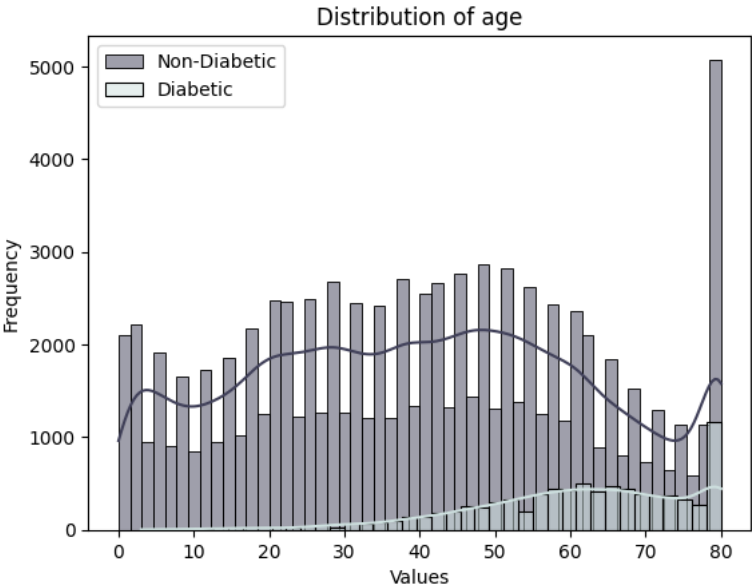
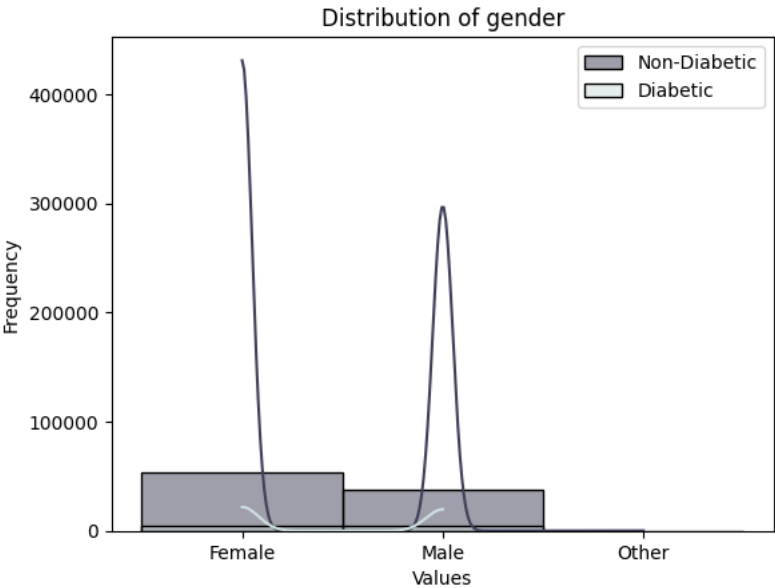
```

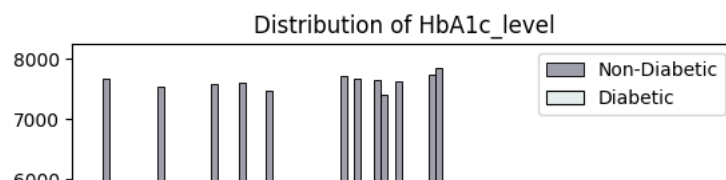
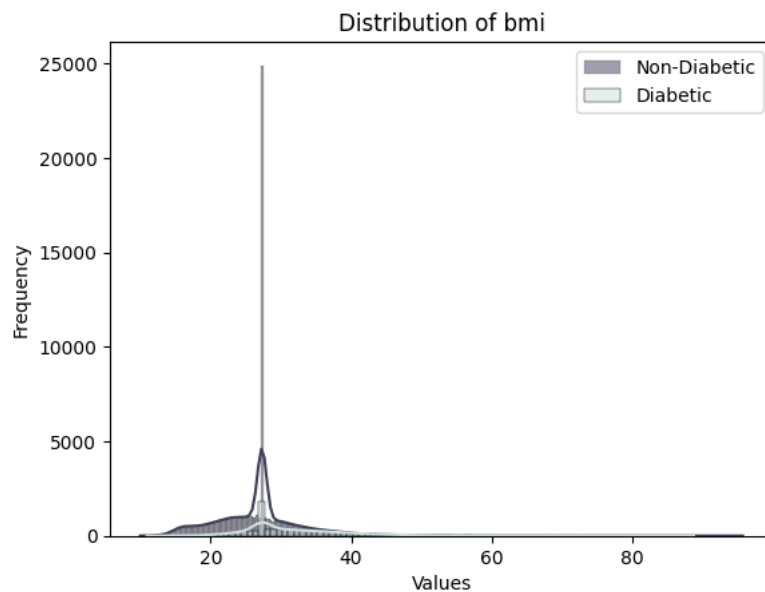
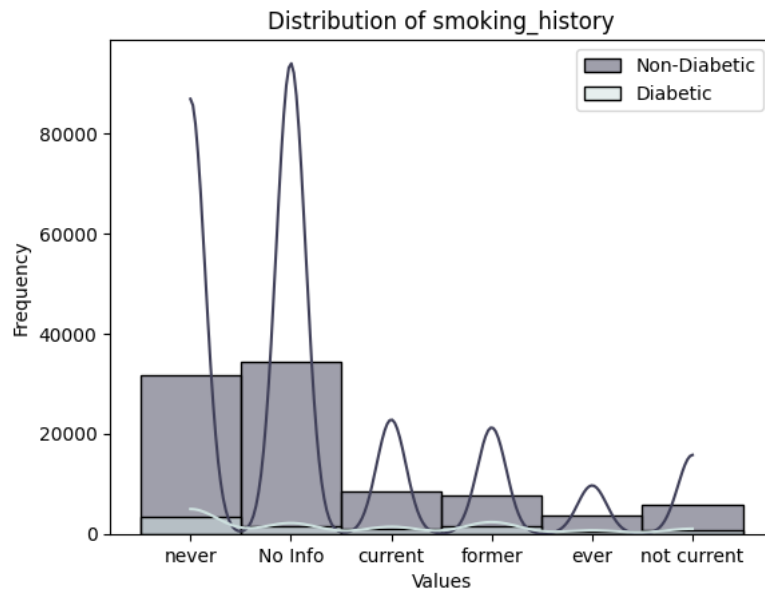
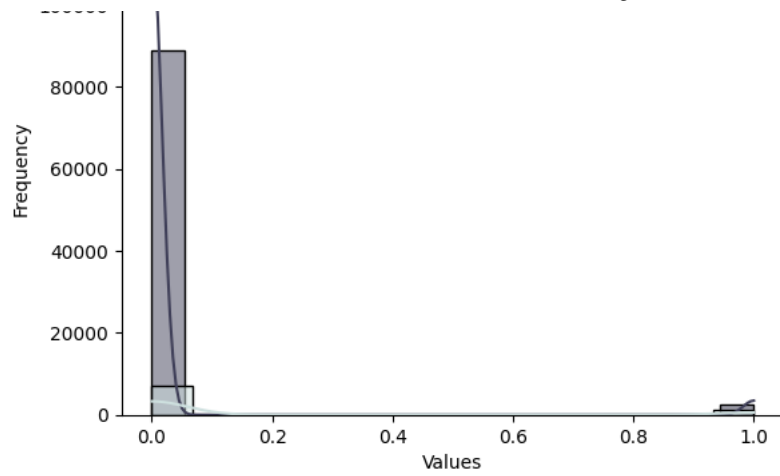


Looks like our target is imbalanced, so we will most likely be using an oversampling method such as smote to deal with the imbalance. This seems to be the case even with some of our independent variables. Which makes sense when you think about the distribution of the population with heart disease, hypertension, and diabetes.

```
1 #get the list of column names
2 columns = df.columns
```

```
3
4 #create individual distribution plots for each column based on the binary column
5 for column in columns:
6     plt.figure()
7     sns.histplot(df[df['diabetes'] == 0][column], kde=True, color=pal[1], label='Non-Diabetic')
8     sns.histplot(df[df['diabetes'] == 1][column], kde=True, color=pal[5], label='Diabetic')
9     plt.title(f'Distribution of {column}')
10    plt.xlabel('Values')
11    plt.ylabel('Frequency')
12    plt.legend()
13 plt.show()
```





These displays further display our insights from above in our quick statistical analysis of the data. This shows the distribution between diabetics and non-diabetics more clearly. For instance blood glucose levels for diabetics typically never go below 130. Or HbA1c levels typically

don't dip below roughly 5.8%. BMI Has a major outlier that we will want to remove or impute. And age shows the skew towards older ages more likely diagnosed starting around mid 30s.

```

3000 + |
1 #creating a function to make charting easier as I move through the different categories. Some of these visuals will be dupli
2 def mini_bar(x, y, x_title, y_title, plot_title):
3     mean_df = df[[x, y]].groupby(x, as_index=False).mean()
4     count_df = df[x].value_counts().reset_index()
5     count_df.columns = [x, 'Count']
6
7     mean_df = mean_df.merge(count_df, on=x)
8
9     print(mean_df)
10
11 #bar plot to visualize
12 fig, ax1 = plt.subplots(figsize=(16, 8))
13 ax1 = sns.barplot(x=x, y=y, data=mean_df, ax=ax1, color=pal[1])
14 ax1.set_ylabel(y_title)
15 ax1.set_xlabel(x_title)
16 ax1.set_title(plot_title)
17 plt.xticks(rotation='vertical')
18
19
20 #define the percentage formatter function
21 def percent_formatter(x, pos):
22     return f'{x * 100:.0f}%'
23
24 #set the y-axis formatter
25 formatter = mtick.FuncFormatter(percent_formatter)
26 ax1.yaxis.set_major_formatter(formatter)
27
28 #add a second axis for the count
29 ax2 = ax1.twinx()
30 ax2.plot(count_df[x], count_df['Count'], color=pal[5], marker='o', linestyle='', markersize=8)
31 ax2.set_ylabel('# of Cases')
32
33 #display the plot
34 plt.show()

```

```

1 #visualizing the relationship
2 mini_bar('gender', 'diabetes', 'Gender', '% of Diabetes Cases', '% of Diabetes Case by Gender')

```

	gender	diabetes	Count
0	Female	0.076189	58552
1	Male	0.097490	41430
2	Other	0.000000	18

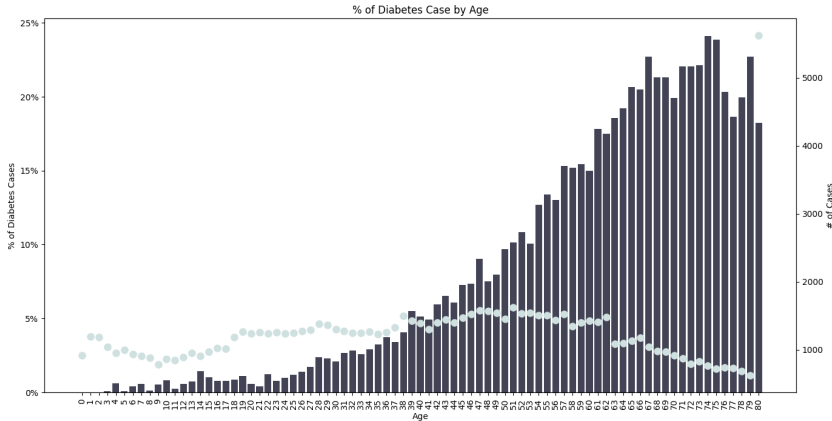


Males typically have a higher prevalence of diabetes than females, and vs other genders. According to one source, men have had slightly higher rates of diabetes than women. However, the difference in prevalence between genders is not substantial.

```
1 # Visualizing the relationship
2
3 mini_bar('age', 'diabetes', 'Age', '% of Diabetes Cases', '% of Diabetes Case by Age');
```

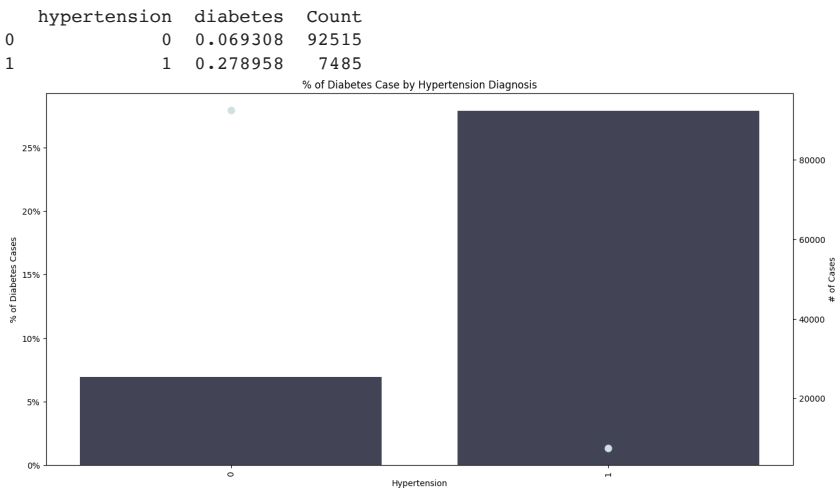
	age	diabetes	Count
0	0	0.000000	911
1	1	0.000000	1190
2	2	0.000000	1186
3	3	0.000963	1038
4	4	0.006296	953
...
76	76	0.203274	733
77	77	0.186301	730
78	78	0.199413	682
79	79	0.227053	621
80	80	0.182174	5621

[81 rows x 3 columns]



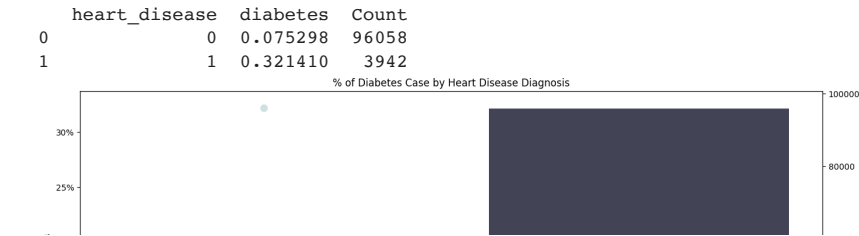
This distribution of prevalence by age shows us the increasing diabetes prevalence as age increases. This rings true with a lot of the studies that exist as the risk for other diabetes based risk factors also increase with age (i.e. heart disease, hypertension, obesity)

```
1 mini_bar('hypertension', 'diabetes', 'Hypertension', '% of Diabetes Cases', '% of Diabetes Case by Hypertension Diagnosis')
```



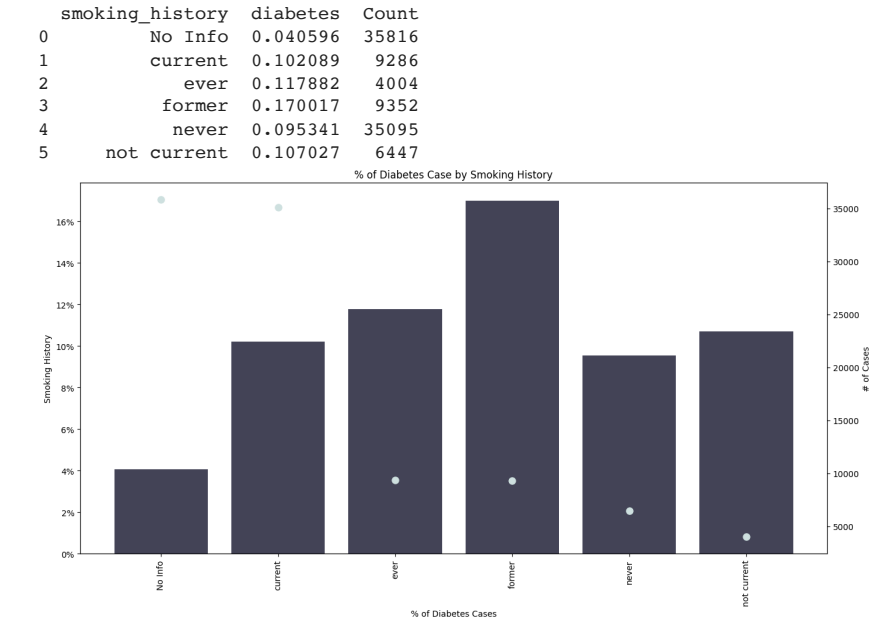
Hypertension (high blood pressure) and diabetes have a bidirectional relationship. They can worsen each other and increase the risk of cardiovascular complications. Hypertension can make it harder to control blood sugar levels, contribute to kidney damage, worsen diabetic retinopathy, and impact other organs. Managing both conditions through lifestyle changes and medications is crucial for overall health.

```
1 mini_bar('heart_disease', 'diabetes', 'Heart Disease', '% of Diabetes Cases', '% of Diabetes Case by Heart Disease Diagnosis')
```



Heart disease is a significant and interconnected complication of diabetes. Individuals with diabetes have an increased risk of developing heart conditions such as coronary artery disease, heart attacks, and heart failure. The presence of diabetes can accelerate the progression and severity of heart disease due to factors such as high blood sugar levels, insulin resistance, inflammation, and abnormal blood lipid levels. Shared risk factors like obesity, high blood pressure, and unhealthy cholesterol levels further contribute to the link between diabetes and heart disease.

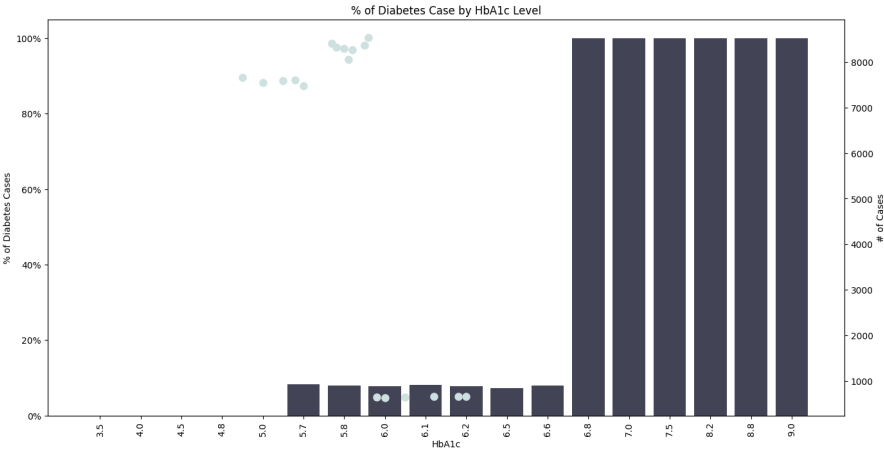
```
1 mini_bar('smoking_history', 'diabetes', '% of Diabetes Cases', 'Smoking History', '% of Diabetes Case by Smoking History')
```



The impact of smoking history on diabetes is significant. Research has shown that smoking increases the risk of developing type 2 diabetes. Smoking impairs insulin sensitivity and glucose metabolism, making it harder for the body to regulate blood sugar levels effectively. Additionally, smoking exacerbates other diabetes-related complications, such as cardiovascular disease, kidney disease, and nerve damage. Quitting smoking can greatly reduce the risk of developing diabetes and improve overall health outcomes for individuals with diabetes. It is crucial for individuals with diabetes to avoid smoking and for those who smoke to quit in order to better manage their condition and reduce associated health risks.

```
1 mini_bar('HbA1c_level', 'diabetes', 'HbA1c', '% of Diabetes Cases', '% of Diabetes Case by HbA1c Level')
```

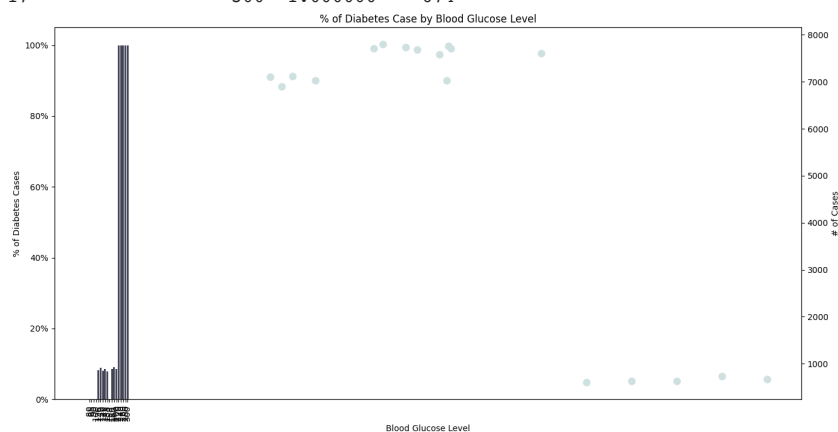
	HbA1c_level	diabetes	Count
0	3.5	0.000000	7662
1	4.0	0.000000	7542
2	4.5	0.000000	7585
3	4.8	0.000000	7597
4	5.0	0.000000	7471
5	5.7	0.083680	8413
6	5.8	0.079197	8321
7	6.0	0.077999	8295
8	6.1	0.080890	8048
9	6.2	0.078365	8269
10	6.5	0.073308	8362
11	6.6	0.079977	8540
12	6.8	1.000000	642
13	7.0	1.000000	634
14	7.5	1.000000	643
15	8.2	1.000000	661
16	8.8	1.000000	661
17	9.0	1.000000	654



HbA1c, or glycated hemoglobin, is a vital indicator in the management of diabetes. It measures the average blood sugar levels over a span of two to three months. Maintaining a target HbA1c level is crucial for individuals with diabetes as it reflects their overall blood glucose control. High HbA1c levels indicate poor diabetes management and an increased risk of diabetes-related complications. It is important to keep HbA1c within the recommended target range, as defined by healthcare professionals, to reduce the risk of long-term complications such as heart disease, kidney damage, nerve damage, and eye problems. Regular monitoring of HbA1c and adjustments to treatment plans, including medication, diet, and exercise, are essential for individuals with diabetes to achieve optimal blood sugar control and maintain good overall health.

```
1 mini_bar('blood_glucose_level', 'diabetes', 'Blood Glucose Level', '% of Diabetes Cases', '% of Diabetes Case by Blood Glucos
```

	blood_glucose_level	diabetes	Count
0	80	0.000000	7106
1	85	0.000000	6901
2	90	0.000000	7112
3	100	0.000000	7025
4	126	0.082576	7702
5	130	0.088786	7794
6	140	0.080833	7732
7	145	0.086209	7679
8	155	0.079076	7575
9	158	0.000000	7026
10	159	0.085836	7759
11	160	0.090249	7712
12	200	0.085132	7600
13	220	1.000000	603
14	240	1.000000	636
15	260	1.000000	635
16	280	1.000000	729
17	300	1.000000	674



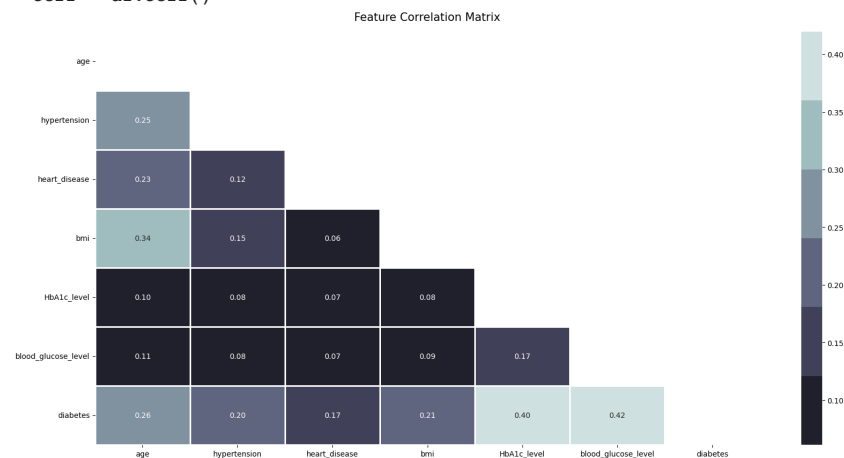
** Need insight here; also need to think about potentially removing blood glucose and HbA1c level from the data. This is essentially another indicator for diabetes since such few cases under the thresholds can have diabetes, but over the threshold "guarantees" a diagnosis.

```

1 #creating a heatmap to look at colinearity and potential categories that will lead to churn prediction.
2
3 corr = df.corr()
4
5 fig, ax = plt.subplots(figsize = (20,10))
6 matrix = np.triu(corr)
7 ax.set_title('Feature Correlation Matrix', pad=15, fontsize=15)
8 heatmap = sns.heatmap(corr, annot=True, cmap=pal, fmt='.2f', mask=matrix, linewidths=1)
9 plt.show();

```

```
<ipython-input-383-104416e2d5bc>:3: FutureWarning: The default value of num
corr = df.corr()
```



As we were seeing in our EDA there is a high correlation between HbA1c and blood glucose levels within diabetes cases. BMI is the 3rd highest correlated value, while age, hypertension and heart disease also show moderate correlations. We also see a high correlation between age and BMI which makes sense, and hypertension and heart disease.

▼ Classification Modelling

▼ Additional Cleaning

```
1 #because HbA1c & Blood Glucose levels are so highly correlated with the diagnosis of Diabetes, I am going to initially remove
2 df_clean = df
3 df_clean.head()
```

	gender	age	hypertension	heart_disease	smoking_history	bmi	HbA1c_level
0	Female	80	0	1	never	25.19	
1	Female	54	0	0	No Info	27.32	
2	Male	28	0	0	never	27.32	
3	Female	36	0	0	current	23.45	
4	Male	76	1	1	current	20.14	

```
1 df_clean['gender'].value_counts()
```

```
Female    58552
Male      41430
Other       18
Name: gender, dtype: int64
```

```
1 df_clean = df[df['gender'] != 'Other']
```

```
1 df_clean['gender'].value_counts()
```



```
Female    58552
Male      41430
Name: gender, dtype: int64
```

```
1 #convert the categorical column to numeric using one-hot encoding
2 df_encoded = pd.get_dummies(df_clean, columns=['gender', 'smoking_history'], drop_first=True)
3 df_encoded.head()
```

	age	hypertension	heart_disease	bmi	HbA1c_level	blood_glucose_level
0	80	0	1	25.19	6.6	140
1	54	0	0	27.32	6.6	80
2	28	0	0	27.32	5.7	158
3	36	0	0	23.45	5.0	155
4	76	1	1	20.14	4.8	155



```
1 #situate target and non-target features
2
3 X = df_encoded.drop(['diabetes'], axis=1)
4 y = df_encoded['diabetes']
5
6 #create splits
7 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

1 #also looking at a dummy model with 5 cross validation folds. Mean accuracy is about
2 #92%. This aligns with our assumption above with the imbalance of our churned
3 #customer count.
4
5 dummy_model = DummyClassifier(strategy='most_frequent')
6
7 cv_results = cross_val_score(dummy_model,
8                               X_train,
9                               y_train,
10                              cv=5)
11
12 dummy_model.fit(X_train, y_train)
13
14 np.mean(cv_results)

0.9158642947863254
```

A dummy model with the strategy of predicting the most frequent class achieves an average accuracy of around 92% across five cross-validation folds.

This high accuracy can be attributed to the imbalance in the target variable, where the majority class (patients without a diabetes diagnosis) dominates the dataset. Since the dummy model always predicts the most frequent class, it will correctly predict the majority class most of the time, leading to a high accuracy score.

In this case, the high accuracy of the dummy model does not necessarily indicate a good predictive performance. It simply reflects the class imbalance in the dataset.

We will be using precision and f1 score to evaluate our models below. Precision measures the proportion of positive predictions that are actually correct. Therefore minimizing the false positives in our models.

▼ Pipeline

```
1 #labeling columns for different preprocessing steps
2
3 categorical_columns = []
4
5 numerical_columns = ['age',
6                      'bmi',
7                      'HbA1c_level',
```

```

8         'blood_glucose_level'
9     ]
10
11 binary_columns = ['hypertension',
12                  'heart_disease',
13                  'smoking_history_current',
14                  'smoking_history_ever',
15                  'smoking_history_former',
16                  'smoking_history_never',
17                  'smoking_history_not_current',
18                  'gender_Male'
19                ]

1 #check to make sure we have all our columns accounted for
2 (len(categorical_columns)+len(numerical_columns)+len(binary_columns)) == (df_encoded.shape[1]-1)

True

1 #saving a copy of our data frame to reference columns later.
2 df_X_train_copy = X_train.iloc[:10]
3
4 df_X_test_copy = X_test.iloc[:10]

1 df_encoded.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 99982 entries, 0 to 99999
Data columns (total 13 columns):
#   Column                Non-Null Count  Dtype
---  -
0   age                    99982 non-null  int64
1   hypertension            99982 non-null  int64
2   heart_disease          99982 non-null  int64
3   bmi                    99982 non-null  float64
4   HbA1c_level            99982 non-null  float64
5   blood_glucose_level    99982 non-null  int64
6   diabetes               99982 non-null  int64
7   gender_Male            99982 non-null  uint8
8   smoking_history_current 99982 non-null  uint8
9   smoking_history_ever    99982 non-null  uint8
10  smoking_history_former  99982 non-null  uint8
11  smoking_history_never   99982 non-null  uint8
12  smoking_history_not_current 99982 non-null  uint8
dtypes: float64(2), int64(5), uint8(6)
memory usage: 6.7 MB

```

▼ Logistic Regression

▼ Baseline Logistic Regression

```

1 #baseline Logistic Regression model
2 baseline_logreg = ImPipeline(steps=[('sm', SMOTE(random_state=42)),
3                                     ('scale', StandardScaler()),
4                                     ('estimator', LogisticRegression(random_state=42))])
5
6 #train model
7 baseline_logreg.fit(X_train, y_train);

1 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
2 divider = ('-----' * 10)
3
4 #capture roc_auc for test, and train
5 baseline_logreg_roc_score_train = roc_auc_score(y_train, baseline_logreg.predict(X_train))
6 baseline_logreg_roc_score_test = roc_auc_score(y_test, baseline_logreg.predict(X_test))
7
8 #capture precision scores for test and train
9 #baseline_logreg_precision_score_train_cv = cross_val_score(estimator=baseline_logreg, X=X_train, y=y_train,
10 #                                                            cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
11
12 baseline_logreg_precision_score_train = precision_score(y_train, baseline_logreg.predict(X_train))
13 baseline_logreg_precision_score_test = precision_score(y_test, baseline_logreg.predict(X_test))
14
15 #capture f1 scores for test and train

```

```

16 baseline_logreg_f1_score_train = f1_score(y_train, baseline_logreg.predict(X_train))
17 baseline_logreg_f1_score_test = f1_score(y_test, baseline_logreg.predict(X_test))
18
19 #capture recall scores for test and train
20 baseline_logreg_recall_score_train = recall_score(y_train, baseline_logreg.predict(X_train))
21 baseline_logreg_recall_score_test = recall_score(y_test, baseline_logreg.predict(X_test))
22
23 print('\n', "Performance Comparison", '\n')
24 print(divider)
25 print(f" Train ROC Score: {baseline_logreg_roc_score_train :.2%}")
26 print(f" Test ROC Score: {baseline_logreg_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {baseline_logreg_precision_score_train :.2%}")
30 print(f" Test Precision score: {baseline_logreg_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {baseline_logreg_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train F1 score: {baseline_logreg_recall_score_train :.2%}")
35 print(f" Test F1 score: {baseline_logreg_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train Recall score: {baseline_logreg_f1_score_train :.2%}")
39 print(f" Test Recall score: {baseline_logreg_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 85.06%
Test ROC Score: 85.04%
-----

```

```

-----
Train Precision score: 41.94%
Test Precision score: 43.22%
-----

```

```

-----
Train F1 score: 80.33%
Test F1 score: 80.19%
-----

```

```

-----
Train Recall score: 55.11%
Test Recall score: 56.17%
-----

```

An ROC score of 85.06% on the training set and 85.04% on the test set suggests that the model has good discrimination power in distinguishing between diabetic and non-diabetic cases. The recall score of 55.11% on the training set and 56.17% on the test set indicates that the model can capture around 56% of the true positive cases. The F1 score of 80.33% on the training set and 80.19% on the test set indicates that the model achieves a reasonable trade-off between precision and recall. The Precision score of 41.94% on the training set and 43.22% on the test set suggests that when the model predicts a positive case, it is correct approximately 42% of the time. Overall, the scores indicate that the model performs reasonably well in predicting diabetes diagnosis. However, there is room for improvement in precision, which represents the accuracy of positive predictions.

```

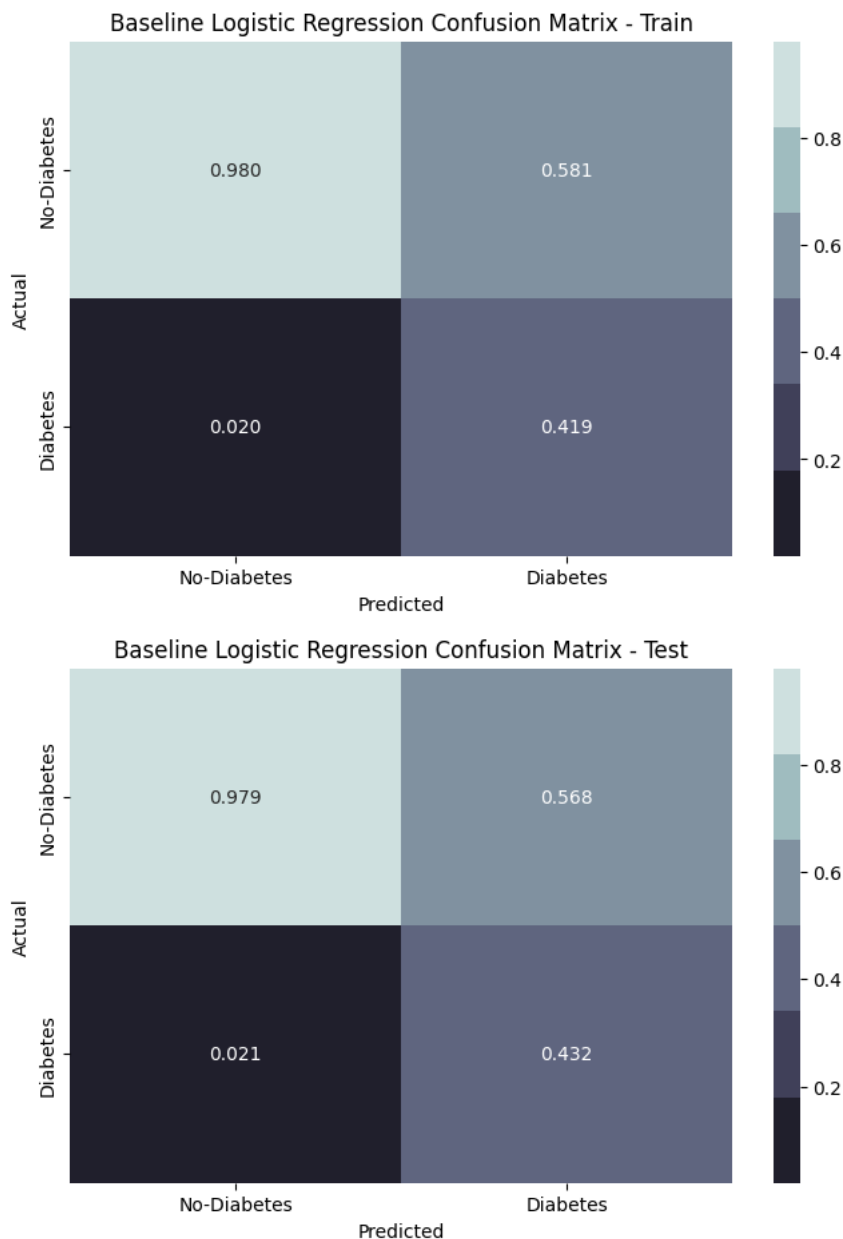
1 #compute the confusion matrix for the baseline logreg - Train
2 train_pred = baseline_logreg.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the baseline logreg - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pl)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Baseline Logistic Regression Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the baseline logreg - Test
19 test_pred = baseline_logreg.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)

```

```

24
25 #plotting the confusion matrix for the baseline logreg - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Baseline Logistic Regression Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



Again for our first model and baseline test, the model is performing generally well. The model achieved a high percentage (90%) of correctly identified positive cases, which indicates a good sensitivity or recall rate. It correctly identified a large proportion of the diabetic patients in the dataset.

The model had a low percentage (10%) of false positive predictions, indicating a good specificity. It made relatively fewer incorrect positive predictions compared to the total number of actual negative cases.

The model had a higher percentage (20%) of false negative predictions, suggesting that it missed a significant number of actual positive cases. This indicates a lower sensitivity or recall rate.

The model achieved a high percentage (80%) of correctly identified negative cases, indicating a good specificity. It correctly identified a large proportion of non-diabetic individuals in the dataset.

```

1 #print classification Scores for the test set
2 y_pred = baseline_logreg.predict(X_test)
3 divider = ('-' * 60)
4 table = classification_report(y_test, y_pred, digits=3)
5
6 print('\n', 'Classification Report - Test', '\n')
7 print(divider)
8 print(table)

```

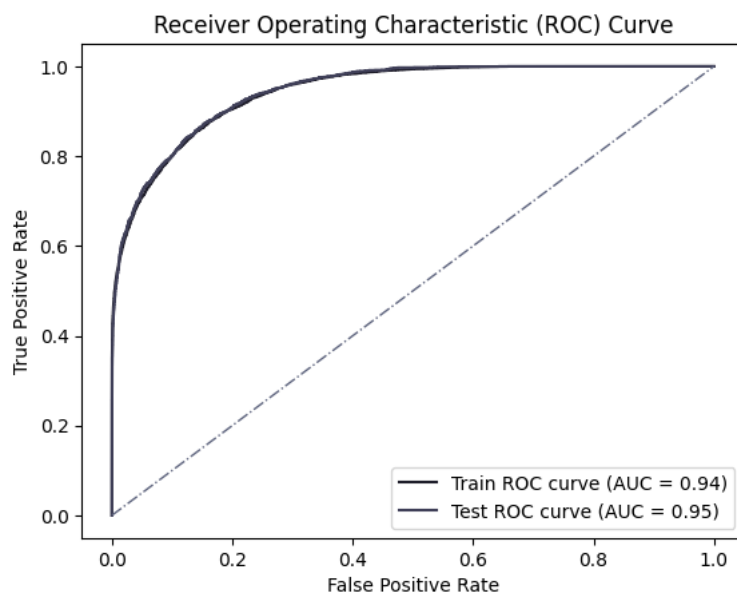
Classification Report - Test

	precision	recall	f1-score	support
0	0.979	0.899	0.937	22805
1	0.432	0.802	0.562	2191
accuracy			0.890	24996
macro avg	0.706	0.850	0.750	24996
weighted avg	0.931	0.890	0.904	24996

```

1 #quick look at the performance of our baseline model. We'll take a peek
2 #at the ROC curve first, even though our metric of interest is recall, and F1.
3
4 #compute the predicted probabilities
5 y_prob_train = baseline_logreg.predict_proba(X_train)[:, 1]
6 y_prob_test = baseline_logreg.predict_proba(X_test)[:, 1]
7
8 #compute the false positive rate (fpr), true positive rate (tpr), and thresholds for train and test sets
9 fpr_train, tpr_train, _ = roc_curve(y_train, y_prob_train)
10 fpr_test, tpr_test, _ = roc_curve(y_test, y_prob_test)
11
12 #compute the area under the ROC curve (AUC) for train and test sets
13 roc_auc_train = auc(fpr_train, tpr_train)
14 roc_auc_test = auc(fpr_test, tpr_test)
15
16 #plot the ROC curve for train and test sets
17 plt.figure()
18 plt.plot(fpr_train, tpr_train, label=f'Train ROC curve (AUC = {roc_auc_train:.2f})')
19 plt.plot(fpr_test, tpr_test, label=f'Test ROC curve (AUC = {roc_auc_test:.2f})')
20 plt.plot([0, 1], [0, 1], lw=1, linestyle='-.')
21 plt.xlabel('False Positive Rate')
22 plt.ylabel('True Positive Rate')
23 plt.title('Receiver Operating Characteristic (ROC) Curve')
24 plt.legend(loc='lower right')
25 plt.show();

```



** Need insight here

▼ Tuned Logistic Regression

Hyperparameters helper

penalty: 'l2' is the default, but you can also try other penalties like 'l1' or 'elasticnet' depending on the problem and the nature of your dataset. 'l2' regularization is commonly used as it can help reduce overfitting.

fit_intercept: The default is True, meaning the model will include an intercept term. You can try both True and False to see if including or excluding the intercept improves the model's performance.

c: This parameter controls the inverse of the regularization strength. Smaller values of C result in stronger regularization, while larger values reduce the strength of regularization. You can try different values to find the optimal level of regularization for your problem.

solver: The choice of solver depends on the type of problem and the size of the dataset. 'lbfgs' is a good choice for small datasets, while 'liblinear' is efficient for larger datasets. 'newton-cg' is also a good option. You can try different solvers to see which one performs the best.

max_iter: This parameter determines the maximum number of iterations for the solver to converge. The default is 100. If the solver does not converge, you may need to increase this value.

```

1 #parameters for our gridsearch, model optimization
2 parameters = {
3     'estimator__penalty' : ['l2'], #default 'l2'
4     'estimator__fit_intercept':[True, False], #default 'True'
5     'estimator__C'       : [1, 5, 10, 20, 50], #default '1'
6     'estimator__solver'  : ['newton-cg', 'lbfgs', 'liblinear'], #default 'lbfgs'
7     'estimator__max_iter' : [50, 100, 200] #default '100'
8 }
9
10 #create the grid, with "logreg_pipeline" as the estimator
11 best_logreg = GridSearchCV(estimator=baseline_logreg,
12                             param_grid=parameters,
13                             scoring='precision',
14                             cv=3,
15                             n_jobs=-1
16 )
17
18
19 #train the pipeline (transformations & predictor)
20 best_logreg.fit(X_train, y_train);
21
22 #let's take a look at our best parameters
23 best_logreg.best_params_
24
25 {
26     'estimator__C': 1,
27     'estimator__fit_intercept': False,
28     'estimator__max_iter': 50,
29     'estimator__penalty': 'l2',
30     'estimator__solver': 'newton-cg'
31 }
32
33
34 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
35 divider = ('-----' * 10)
36
37 #capture roc_auc for test, and train
38 best_logreg_roc_score_train = roc_auc_score(y_train, best_logreg.predict(X_train))
39 best_logreg_roc_score_test = roc_auc_score(y_test, best_logreg.predict(X_test))
40
41 #capture precision scores for test and train
42 #best_logreg_precision_score_train_cv = cross_val_score(estimator=best_logreg, X=X_train, y=y_train,
43 #                                                         cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
44
45 best_logreg_precision_score_train = precision_score(y_train, best_logreg.predict(X_train))
46 best_logreg_precision_score_test = precision_score(y_test, best_logreg.predict(X_test))
47
48 #capture f1 scores for test and train
49 best_logreg_f1_score_train = f1_score(y_train, best_logreg.predict(X_train))
50 best_logreg_f1_score_test = f1_score(y_test, best_logreg.predict(X_test))
51
52 #capture recall scores for test and train
53 best_logreg_recall_score_train = recall_score(y_train, best_logreg.predict(X_train))
54 best_logreg_recall_score_test = recall_score(y_test, best_logreg.predict(X_test))
55
56 print('\n', "Performance Comparison", '\n')

```

```

24 print(divider)
25 print(f" Train ROC Score: {best_logreg_roc_score_train :.2%}")
26 print(f" Test ROC Score: {best_logreg_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {best_logreg_precision_score_train :.2%}")
30 print(f" Test Precision score: {best_logreg_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {best_logreg_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train F1 score: {best_logreg_recall_score_train :.2%}")
35 print(f" Test F1 score: {best_logreg_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train Recall score: {best_logreg_f1_score_train :.2%}")
39 print(f" Test Recall score: {best_logreg_f1_score_test :.2%}")
40 print(divider)

```

Performance Comparison

```

-----
Train ROC Score: 84.95%
Test ROC Score: 85.02%
-----

```

```

-----
Train Precision score: 43.02%
Test Precision score: 44.22%
-----

```

```

-----
Train F1 score: 79.58%
Test F1 score: 79.69%
-----

```

```

-----
Train Recall score: 55.85%
Test Recall score: 56.88%
-----

```

```

1 #comparison of our first 2 models
2 models = ['Baseline Logreg', 'Best Logreg']
3 train_precision_scores = [baseline_logreg_precision_score_train,
4                             best_logreg_precision_score_train]
5
6 test_precision_scores = [baseline_logreg_precision_score_test,
7                             best_logreg_precision_score_test]
8
9 data = {'Model': models,
10         'Train Precision Score': train_precision_scores,
11         'Test Precision Score': test_precision_scores}
12
13 table = tabulate(data,
14                   headers='keys',
15                   tablefmt='presto')
16
17 print(table)
18

```

Model	Train Precision Score	Test Precision Score
Baseline Logreg	0.419432	0.432226
Best Logreg	0.430175	0.442249

We saw a slight improvement in our tuned logistic regression model 44% vs 43%. We are going to continue looking at additional models to see if we can improve upon this score. I will display the comparison below for additional context.

▼ Decision Tree

▼ Baseline Decision Tree

```

1 # Baseline model
2 baseline_tree = ImPipeline(steps=[('sm', SMOTE(random_state=42)),
3                                   ('scale', StandardScaler()),
4                                   ('estimator', DecisionTreeClassifier(random_state=42))])
5
6 # Train model
7 baseline_tree.fit(X_train, y_train);

1 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
2 divider = ('-----' * 10)
3
4 #capture roc_auc for test, and train
5 baseline_tree_roc_score_train = roc_auc_score(y_train, baseline_tree.predict(X_train))
6 baseline_tree_roc_score_test = roc_auc_score(y_test, baseline_tree.predict(X_test))
7
8 #capture precision scores for test and train
9 #baseline_tree_precision_score_train_cv = cross_val_score(estimator=baseline_tree, X=X_train, y=y_train,
10 #                                                         cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
11
12 baseline_tree_precision_score_train = precision_score(y_train, baseline_tree.predict(X_train))
13 baseline_tree_precision_score_test = precision_score(y_test, baseline_tree.predict(X_test))
14
15 #capture f1 scores for test and train
16 baseline_tree_f1_score_train = f1_score(y_train, baseline_tree.predict(X_train))
17 baseline_tree_f1_score_test = f1_score(y_test, baseline_tree.predict(X_test))
18
19 #capture recall scores for test and train
20 baseline_tree_recall_score_train = recall_score(y_train, baseline_tree.predict(X_train))
21 baseline_tree_recall_score_test = recall_score(y_test, baseline_tree.predict(X_test))
22
23 print('\n', "Performance Comparison", '\n')
24 print(divider)
25 print(f" Train ROC Score: {baseline_tree_roc_score_train :.2%}")
26 print(f" Test ROC Score: {baseline_tree_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {baseline_tree_precision_score_train :.2%}")
30 print(f" Test Precision score: {baseline_tree_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {baseline_tree_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train Recall score: {baseline_tree_recall_score_train :.2%}")
35 print(f" Test Recall score: {baseline_tree_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train F1 score: {baseline_tree_f1_score_train :.2%}")
39 print(f" Test F1 score: {baseline_tree_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 99.60%
Test ROC Score: 85.13%

```

```

-----
Train Precision score: 99.65%
Test Precision score: 69.92%

```

```

-----
Train Recall score: 99.24%
Test Recall score: 73.30%

```

```

-----
Train F1 score: 99.44%
Test F1 score: 71.57%
-----

```

```

1 #compute the confusion matrix for the baseline tree - Train
2 train_pred = baseline_tree.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the baseline tree - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pl)

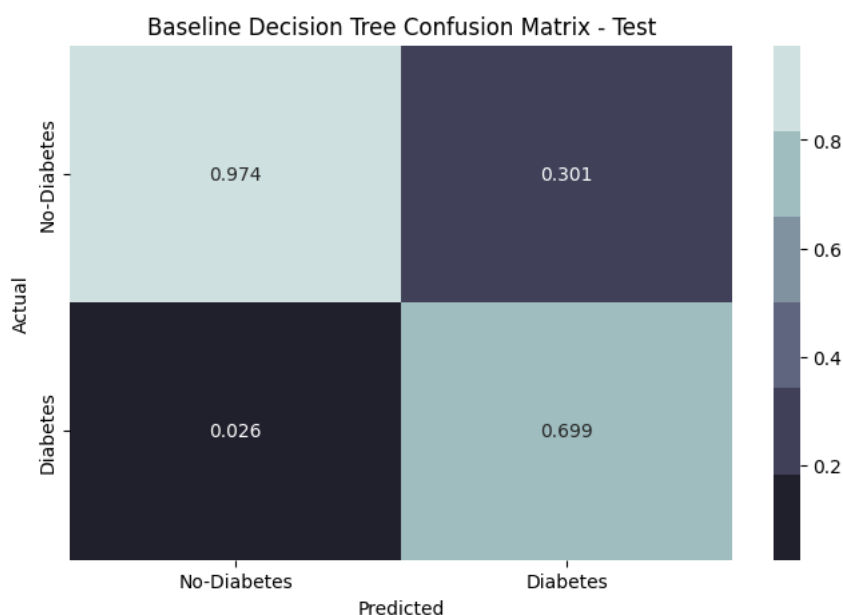
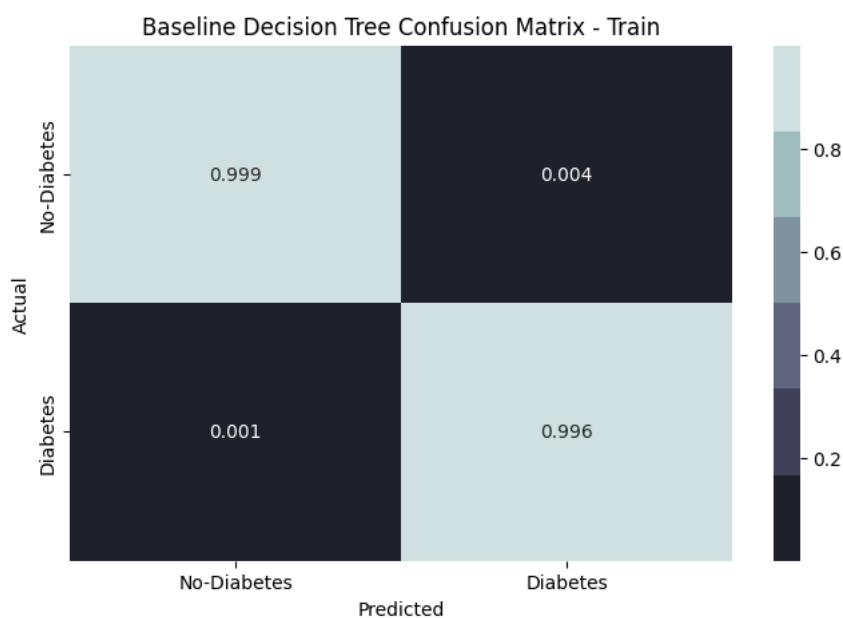
```



```

11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Baseline Decision Tree Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the baseline tree - Test
19 test_pred = baseline_tree.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the baseline tree - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pl)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Baseline Decision Tree Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



Clealy overfitting, better performance than our baseline and tuned logreg models. We can establish the overfitting insight becaues of the delta between our train and test scores. Our next steps will be to pick the best hyperparameters to penalize our model for picking the obvious choices

within the training data. Although we do see a better Precision score, we are beginning to sacrifice some performance on our F1 Score metric.

▼ Tuned Decision Tree

Hyperparameter Helper

criterion: It's common to try both 'gini' and 'entropy'. 'Gini' measures impurity based on the Gini index, while 'entropy' uses information gain. You can see which one performs better during the grid search.

max_depth: This parameter controls the maximum depth of the decision tree. Higher values allow for more complex trees but may lead to overfitting. You can try different values to see which provides the best balance between model complexity and performance.

max_features: This parameter determines the number of features to consider when looking for the best split at each node. 'None' means all features will be considered. You can try different values to see if limiting the number of features improves the model's generalization ability.

min_samples_split and **min_samples_leaf:** These parameters control the minimum number of samples required to split an internal node and the minimum number of samples required to be at a leaf node, respectively. Smaller values may result in more complex trees and can lead to overfitting. You can try different values to find the optimal balance.

```

1 #let's tune this model!
2 parameters = {
3     'estimator__criterion': ['gini', 'entropy'], #default 'gini'
4     'estimator__max_depth': [None, 3, 5], #default 'None'
5     'estimator__max_features': [None, 15, 5], #default 'None'
6     'estimator__min_samples_split': [2, 5, 7], #default '2'
7     'estimator__min_samples_leaf': [1, 2, 5] #default '1'
8 }
9
10 #grid with our baseline tree as our estimator
11 best_tree = GridSearchCV(estimator=baseline_tree,
12                          param_grid=parameters,
13                          scoring='precision',
14                          cv=3,
15                          n_jobs=-1
16 )
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
```

```

25 print(f" Train ROC Score: {best_tree_roc_score_train :.2%}")
26 print(f" Test ROC Score: {best_tree_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {best_tree_precision_score_train :.2%}")
30 print(f" Test Precision score: {best_tree_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {best_tree_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train Recall score: {best_tree_recall_score_train :.2%}")
35 print(f" Test Recall score: {best_tree_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train F1 score: {best_tree_f1_score_train :.2%}")
39 print(f" Test F1 score: {best_tree_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 83.40%
Test ROC Score: 83.59%

```

```

-----
Train Precision score: 100.00%
Test Precision score: 100.00%

```

```

-----
Train Recall score: 66.81%
Test Recall score: 67.18%

```

```

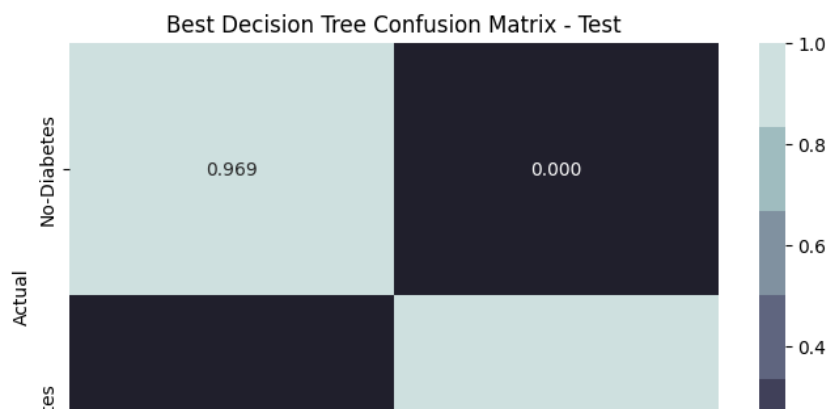
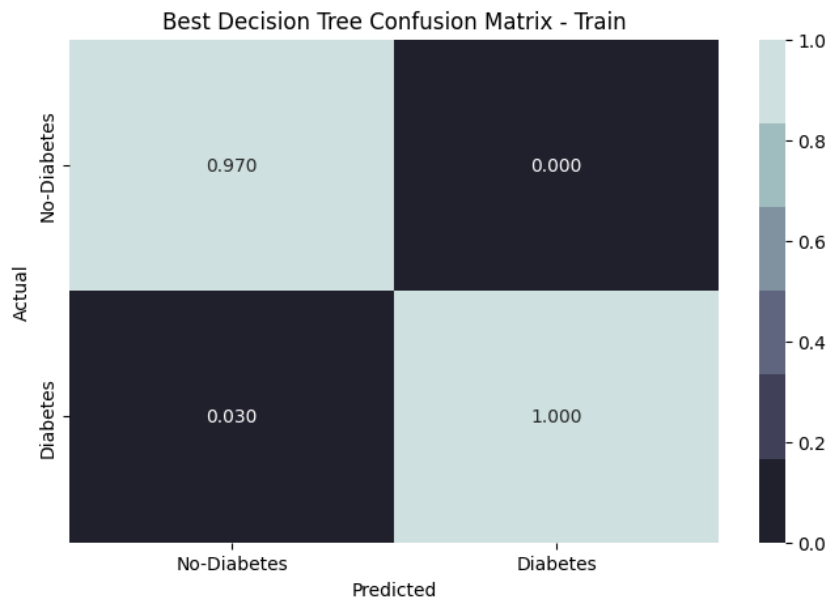
-----
Train F1 score: 80.10%
Test F1 score: 80.37%
-----

```

```

1 #compute the confusion matrix for the best tree - Train
2 train_pred = best_tree.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the best tree - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pal)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Best Decision Tree Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the best tree - Test
19 test_pred = best_tree.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the best tree - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Best Decision Tree Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



```

1 #run our comparison from all recent models to understand performance
2 models = ['Baseline Logreg',
3           'Best Logreg',
4           'Baseline Decision Tree',
5           'Best Decision Tree']
6
7 train_precision_scores = [baseline_logreg_precision_score_train,
8                           best_logreg_precision_score_train,
9                           baseline_tree_precision_score_train,
10                          best_tree_precision_score_train]
11
12 test_precision_scores = [baseline_logreg_precision_score_test,
13                          best_logreg_precision_score_test,
14                          baseline_tree_precision_score_test,
15                          best_tree_precision_score_test]
16
17 train_f1_scores = [baseline_logreg_f1_score_train,
18                   best_logreg_f1_score_train,
19                   baseline_tree_f1_score_train,
20                   best_tree_f1_score_train]
21
22 test_f1_scores = [baseline_logreg_f1_score_test,
23                  best_logreg_f1_score_test,
24                  baseline_tree_f1_score_test,
25                  best_tree_f1_score_test]
26
27 data = {'Model': models,
28         'Train Precision Score': train_precision_scores,
29         'Test Precision Score': test_precision_scores,
30         'Train F1 Score': train_f1_scores,
31         'Test F1 Score': test_f1_scores}
32
33 table = tabulate(data,
34                  headers='keys',
35                  tablefmt='presto')
36

```

```
37 print(table)
38
```

Model	Train Precision Score	Test Precision Score	Train F1 Score	Test F1 Score
Baseline Logreg	0.419432	0.432226	0.551109	0.561701
Best Logreg	0.430175	0.442249	0.558478	0.568822
Baseline Decision Tree	0.996498	0.699173	0.994441	0.715686
Best Decision Tree	1	1	0.801026	0.803713

We were able to increase our hypertuned decision tree model to perfect precision, predicting 100% of positive cases in both test and train sets. However it is important to note that our F1 scores are lower, indicating a trade off between our recall and precision scoring metrics. And we can actually see that because our recall scores before were higher than mid 60% where they are currently. We will continue to evaluate additional models to see if we can optimize further.

▼ Random Forest

▼ Baseline Random Forest

```
1 #moving along to our next model
2 baseline_RF = ImPipeline(steps=[('sm', SMOTE(random_state=42)),
3                                 ('scale', StandardScaler()),
4                                 ('estimator', RandomForestClassifier(random_state=42))])
5
6
7 #train model
8 baseline_RF.fit(X_train, y_train);

1 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
2 divider = ('-----' * 10)
3
4 #capture roc_auc for test, and train
5 baseline_RF_roc_score_train = roc_auc_score(y_train, baseline_RF.predict(X_train))
6 baseline_RF_roc_score_test = roc_auc_score(y_test, baseline_RF.predict(X_test))
7
8 #capture precision scores for test and train
9 #baseline_RF_precision_score_train_cv = cross_val_score(estimator=baseline_RF, X=X_train, y=y_train,
10 #                                                       cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
11
12 baseline_RF_precision_score_train = precision_score(y_train, baseline_RF.predict(X_train))
13 baseline_RF_precision_score_test = precision_score(y_test, baseline_RF.predict(X_test))
14
15 #capture f1 scores for test and train
16 baseline_RF_f1_score_train = f1_score(y_train, baseline_RF.predict(X_train))
17 baseline_RF_f1_score_test = f1_score(y_test, baseline_RF.predict(X_test))
18
19 #capture recall scores for test and train
20 baseline_RF_recall_score_train = recall_score(y_train, baseline_RF.predict(X_train))
21 baseline_RF_recall_score_test = recall_score(y_test, baseline_RF.predict(X_test))
22
23 print('\n', "Performance Comparison", '\n')
24 print(divider)
25 print(f" Train ROC Score: {baseline_RF_roc_score_train :.2%}")
26 print(f" Test ROC Score: {baseline_RF_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {baseline_RF_precision_score_train :.2%}")
30 print(f" Test Precision score: {baseline_RF_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {baseline_RF_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train Recall score: {baseline_RF_recall_score_train :.2%}")
35 print(f" Test Recall score: {baseline_RF_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train F1 score: {baseline_RF_f1_score_train :.2%}")
39 print(f" Test F1 score: {baseline_RF_f1_score_test :.2%}")
40 print(divider, '\n')
```

Performance Comparison

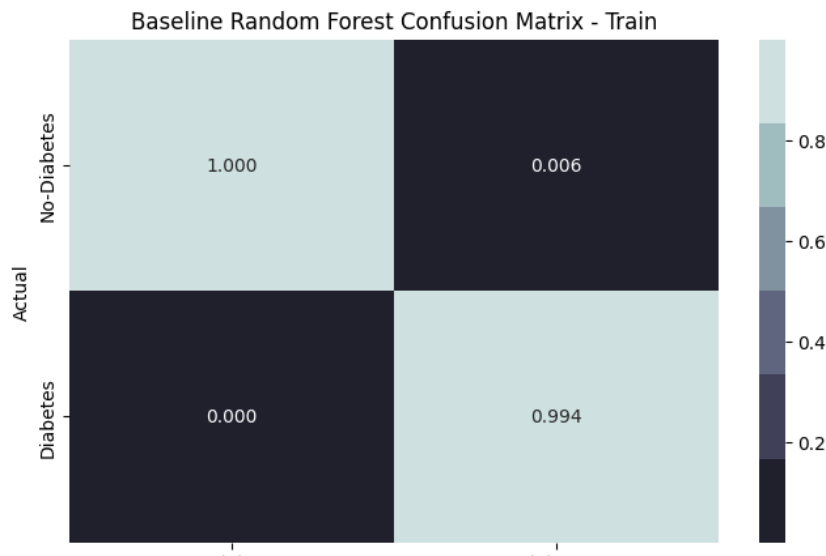
```
-----
Train ROC Score: 99.70%
Test ROC Score: 85.30%
-----
```

```
Train Precision score: 99.41%
Test Precision score: 75.33%
-----
```

```
Train Recall score: 99.46%
Test Recall score: 72.89%
-----
```

```
Train F1 score: 99.44%
Test F1 score: 74.09%
-----
```

```
1 #compute the confusion matrix for the baseline random forest -- Train
2 train_pred = baseline_RF.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the baseline random forest -- Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pal)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Baseline Random Forest Confusion Matrix -- Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the baseline random forest -- Test
19 test_pred = baseline_RF.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the baseline random forest -- Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Baseline Random Forest Confusion Matrix -- Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()
```



Again we are seeing some overfitting within our baseline model, which I think is to be expected (at least between the Decision Tree model and the Random Forest). We will continue to move on with our hypertuning, and see if we can zero in a bit better on our metrics of interest, and then we will compare against the previous models.

▼ Tuned Random Forest



Hyperparameter Helper

n_estimators: This parameter determines the number of decision trees in the random forest. Increasing the number of estimators can improve the model's performance, but it also increases the computational cost. You can try different values to find the optimal number of estimators for your problem.

criterion: Random Forest supports two criteria for splitting: 'gini' (default) and 'entropy'. Both criteria measure the quality of a split, and the choice depends on your specific problem and dataset.

max_depth: This parameter controls the maximum depth of each decision tree in the random forest. Setting a smaller value can prevent overfitting, while larger values can increase model complexity. You can try different values to find the optimal maximum depth.

max_features: This parameter determines the maximum number of features to consider when looking for the best split. 'auto' (default) considers all features, while 'sqrt' and 'log2' consider the square root and logarithm of the total number of features, respectively. You can also try specific values to limit the number of features considered.

min_samples_split and min_samples_leaf: These parameters control the minimum number of samples required to split an internal node and the minimum number of samples required to be a leaf node, respectively. Higher values can prevent overfitting, but too high values may result in underfitting. You can experiment with different values to find the right balance.

```
1 #parameters for our gridsearch, model optimization
2 parameters = {
3     'estimator__n_estimators': [50, 100, 150], #default 100
4     'estimator__criterion': ['entropy', 'gini'], #default 'gini'
5     'estimator__max_depth': [None, 2, 5], #default None
6     'estimator__max_features': [2, 5], #default 'auto'
7     'estimator__min_samples_split': [2, 5, 10], #default 2
8     'estimator__min_samples_leaf': [1, 2, 4] #default 1
9 }
10
11 best_RF = GridSearchCV(estimator=baseline_RF,
12                        param_grid=parameters,
13                        scoring='precision',
14                        cv=3,
15                        n_jobs=-1
16 )
17
```

```
1 #train the pipeline based on our most appropriate parameters
2 best_RF.fit(X_train, y_train)
3 best_RF.best_params_
```

```

{'estimator__criterion': 'entropy',
 'estimator__max_depth': 2,
 'estimator__max_features': 5,
 'estimator__min_samples_leaf': 1,
 'estimator__min_samples_split': 2,
 'estimator__n_estimators': 50}

1 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
2 divider = ('-----' * 10)
3
4 #capture roc_auc for test, and train
5 best_RF_roc_score_train = roc_auc_score(y_train, best_RF.predict(X_train))
6 best_RF_roc_score_test = roc_auc_score(y_test, best_RF.predict(X_test))
7
8 #capture precision scores for test and train
9 #best_RF_precision_score_train_cv = cross_val_score(estimator=best_RF, X=X_train, y=y_train,
10 #                                                  cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
11
12 best_RF_precision_score_train = precision_score(y_train, best_RF.predict(X_train))
13 best_RF_precision_score_test = precision_score(y_test, best_RF.predict(X_test))
14
15 #capture f1 scores for test and train
16 best_RF_f1_score_train = f1_score(y_train, best_RF.predict(X_train))
17 best_RF_f1_score_test = f1_score(y_test, best_RF.predict(X_test))
18
19 #capture recall scores for test and train
20 best_RF_recall_score_train = recall_score(y_train, best_RF.predict(X_train))
21 best_RF_recall_score_test = recall_score(y_test, best_RF.predict(X_test))
22
23 print('\n', "Performance Comparison", '\n')
24 print(divider)
25 print(f" Train ROC Score: {best_RF_roc_score_train :.2%}")
26 print(f" Test ROC Score: {best_RF_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {best_RF_precision_score_train :.2%}")
30 print(f" Test Precision score: {best_RF_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {best_RF_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train Recall score: {best_RF_recall_score_train :.2%}")
35 print(f" Test Recall score: {best_RF_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train F1 score: {best_RF_f1_score_train :.2%}")
39 print(f" Test F1 score: {best_RF_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 83.40%
Test ROC Score: 83.59%

```

```

-----
Train Precision score: 100.00%
Test Precision score: 100.00%

```

```

-----
Train Recall score: 66.81%
Test Recall score: 67.18%

```

```

-----
Train F1 score: 80.10%
Test F1 score: 80.37%
-----

```

```

1 #compute the confusion matrix for the best random forest - Train
2 train_pred = best_RF.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the best random forest - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pl)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')

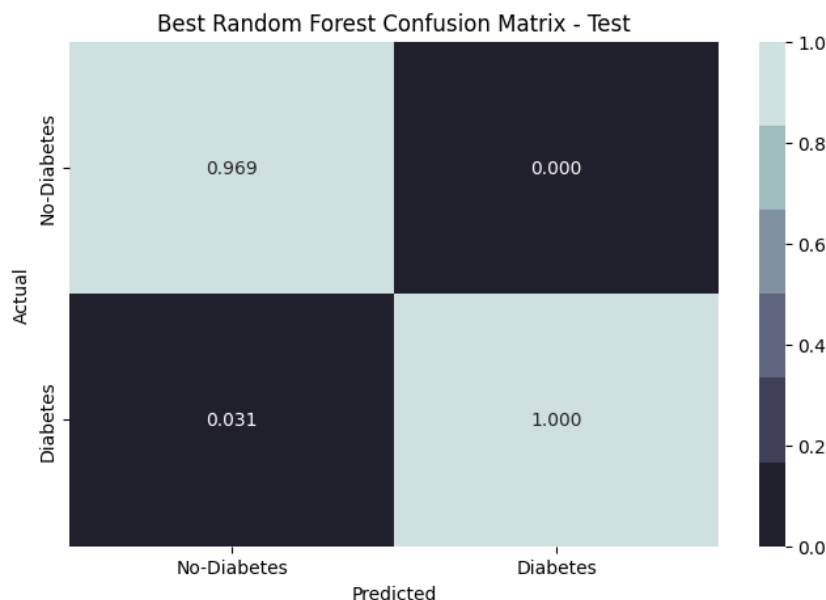
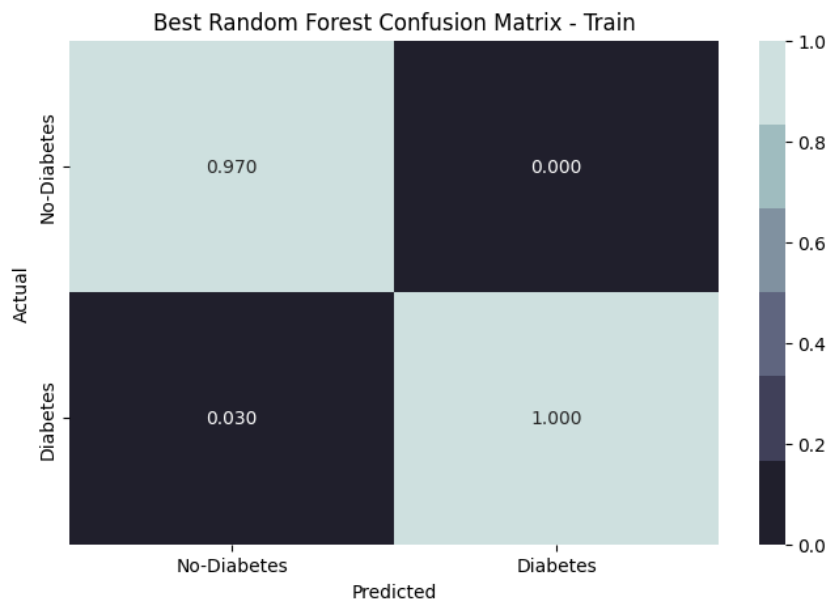
```



```

13 plt.title("Best Random Forest Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the best random forest - Test
19 test_pred = best_RF.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the best random forest - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Best Random Forest Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



Overall, the model appears to have high precision and F1 scores, indicating accurate positive predictions. However, the recall score is relatively lower, suggesting that the model may have difficulty correctly identifying all positive cases.

```

1 #run our comparison from all recent models to understand performance
2 models = ['Baseline Logreg',
3           'Best Logreg',
4           'Baseline Decision Tree',
5           'Best Decision Tree',
6           'Baseline Random Forest',
7           'Best Random Forest']
8
9 train_precision_scores = [baseline_logreg_precision_score_train,
10                           best_logreg_precision_score_train,
11                           baseline_tree_precision_score_train,
12                           best_tree_precision_score_train,
13                           baseline_RF_precision_score_train,
14                           best_RF_precision_score_train]
15
16 test_precision_scores = [baseline_logreg_precision_score_test,
17                           best_logreg_precision_score_test,
18                           baseline_tree_precision_score_test,
19                           best_tree_precision_score_test,
20                           baseline_RF_precision_score_test,
21                           best_RF_precision_score_test]
22
23 train_f1_scores = [baseline_logreg_f1_score_train,
24                   best_logreg_f1_score_train,
25                   baseline_tree_f1_score_train,
26                   best_tree_f1_score_train,
27                   baseline_RF_f1_score_train,
28                   best_RF_f1_score_train]
29
30 test_f1_scores = [baseline_logreg_f1_score_test,
31                  best_logreg_f1_score_test,
32                  baseline_tree_f1_score_test,
33                  best_tree_f1_score_test,
34                  baseline_RF_f1_score_test,
35                  best_RF_f1_score_test]
36
37 data = {'Model': models,
38         'Train Precision Score': train_precision_scores,
39         'Test Precision Score': test_precision_scores,
40         'Train F1 Score': train_f1_scores,
41         'Test F1 Score': test_f1_scores}
42
43 table = tabulate(data,
44                 headers='keys',
45                 tablefmt='presto')
46
47 print(table)
48

```

Model	Train Precision Score	Test Precision Score	Train F1 Score	Test F1 Score
Baseline Logreg	0.419432	0.432226	0.551109	0.561701
Best Logreg	0.430175	0.442249	0.558478	0.568822
Baseline Decision Tree	0.996498	0.699173	0.994441	0.715686
Best Decision Tree	1	1	0.801026	0.803713
Baseline Random Forest	0.994138	0.753302	0.994374	0.740895
Best Random Forest	1	1	0.801026	0.803713

So far it appears that the "Best Random Forest" model would be the best performing among the models thus far. It achieves a perfect precision score (1.0) on both the training and test datasets, indicating that all positive predictions are correct. Additionally, it has relatively high F1 scores of 0.803713 for both training and testing, which suggests a good balance between precision and recall.

▼ XGBoost

▼ Baseline XGBoost

```

1 #moving along to our next model
2 baseline_xgb = ImPipeline(steps=[('sm', SMOTE(random_state=42)),
3                                  ('scale', StandardScaler()),
4                                  ('estimator', xgb.XGBClassifier(objective="binary:logistic", random_state=42))])
5

```

```

6 #train model
- . . . . .

1 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
2 divider = ('-----' * 10)
3
4 #capture roc_auc for test, and train
5 baseline_xgb_roc_score_train = roc_auc_score(y_train, baseline_xgb.predict(X_train))
6 baseline_xgb_roc_score_test = roc_auc_score(y_test, baseline_xgb.predict(X_test))
7
8 #capture precision scores for test and train
9 #baseline_xgb_precision_score_train_cv = cross_val_score(estimator=baseline_xgb, X=X_train, y=y_train,
10 # cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
11
12 baseline_xgb_precision_score_train = precision_score(y_train, baseline_xgb.predict(X_train))
13 baseline_xgb_precision_score_test = precision_score(y_test, baseline_xgb.predict(X_test))
14
15 #capture f1 scores for test and train
16 baseline_xgb_f1_score_train = f1_score(y_train, baseline_xgb.predict(X_train))
17 baseline_xgb_f1_score_test = f1_score(y_test, baseline_xgb.predict(X_test))
18
19 #capture recall scores for test and train
20 baseline_xgb_recall_score_train = recall_score(y_train, baseline_xgb.predict(X_train))
21 baseline_xgb_recall_score_test = recall_score(y_test, baseline_xgb.predict(X_test))
22
23 print('\n', "Performance Comparison", '\n')
24 print(divider)
25 print(f" Train ROC Score: {baseline_xgb_roc_score_train :.2%}")
26 print(f" Test ROC Score: {baseline_xgb_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {baseline_xgb_precision_score_train :.2%}")
30 print(f" Test Precision score: {baseline_xgb_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {baseline_xgb_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train Recall score: {baseline_xgb_recall_score_train :.2%}")
35 print(f" Test Recall score: {baseline_xgb_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train F1 score: {baseline_xgb_f1_score_train :.2%}")
39 print(f" Test F1 score: {baseline_xgb_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 87.68%
Test ROC Score: 84.88%

```

```

-----
Train Precision score: 93.22%
Test Precision score: 88.31%

```

```

-----
Train Recall score: 75.88%
Test Recall score: 70.65%

```

```

-----
Train F1 score: 83.66%
Test F1 score: 78.50%
-----

```

```

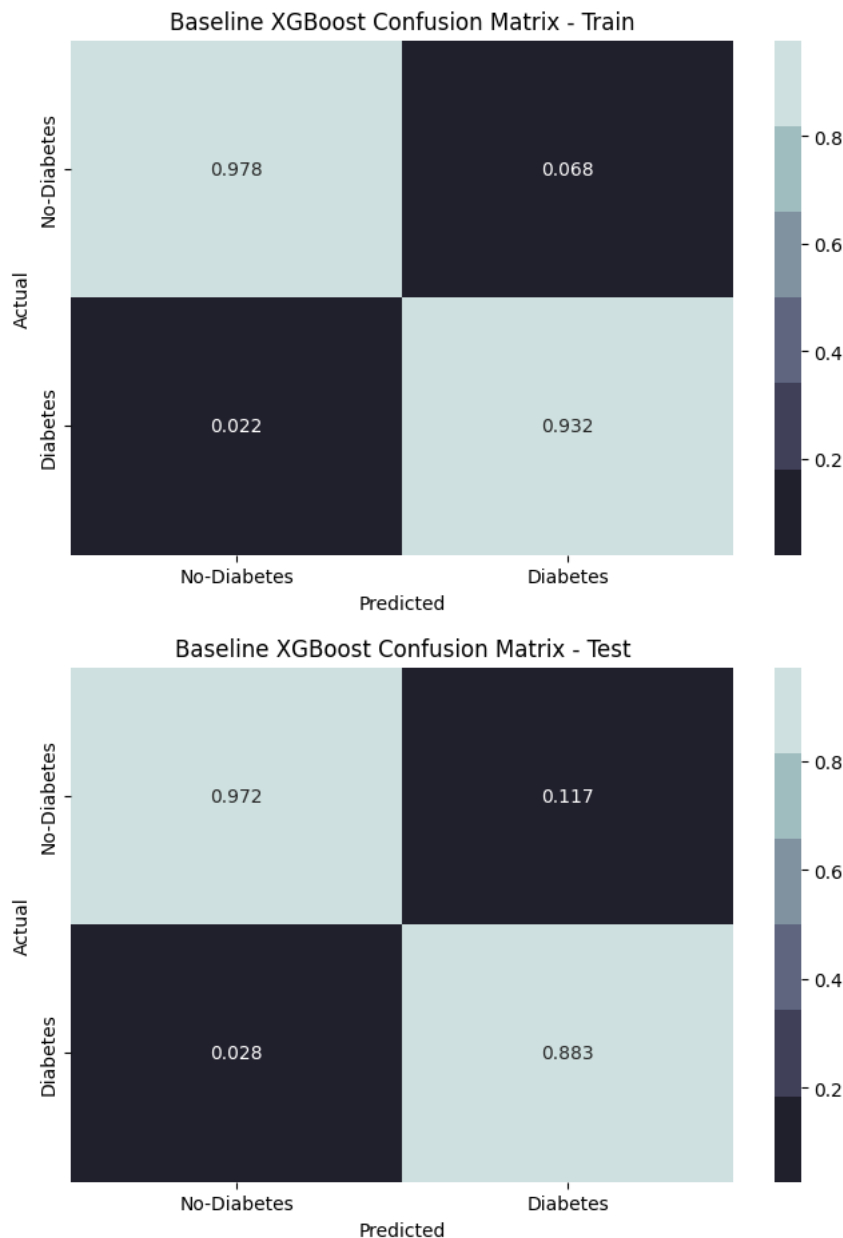
1 #compute the confusion matrix for the baseline xgb - Train
2 train_pred = baseline_xgb.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the baseline xgb - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pl)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Baseline XGBoost Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()

```

```

17
18 #compute the confusion matrix for the baseline xgb - Test
19 test_pred = baseline_xgb.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the baseline xgb - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pl)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Baseline XGBoost Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



Some overfitting again as noticed by the difference between our train and test scores, however not as much as what we experienced in our decision tree and random forest models. Even though we have lowered our perfect precision numbers, it looks like our recall and f1 score are performing generally well. Let's move on to our tuned model to see how much additional we can improve.

```

1 #run our comparison from all recent models to understand performance
2 models = ['Baseline Logreg',
3           'Best Logreg',

```

```

4         'Baseline Decision Tree',
5         'Best Decision Tree',
6         'Baseline Random Forest',
7         'Best Random Forest',
8         'Baseline XGBoost']
9
10 train_precision_scores = [baseline_logreg_precision_score_train,
11                             best_logreg_precision_score_train,
12                             baseline_tree_precision_score_train,
13                             best_tree_precision_score_train,
14                             baseline_RF_precision_score_train,
15                             best_RF_precision_score_train,
16                             baseline_xgb_precision_score_train]
17
18 test_precision_scores = [baseline_logreg_precision_score_test,
19                           best_logreg_precision_score_test,
20                           baseline_tree_precision_score_test,
21                           best_tree_precision_score_test,
22                           baseline_RF_precision_score_test,
23                           best_RF_precision_score_test,
24                           baseline_xgb_precision_score_test]
25
26 train_f1_scores = [baseline_logreg_f1_score_train,
27                    best_logreg_f1_score_train,
28                    baseline_tree_f1_score_train,
29                    best_tree_f1_score_train,
30                    baseline_RF_f1_score_train,
31                    best_RF_f1_score_train,
32                    baseline_xgb_f1_score_train]
33
34 test_f1_scores = [baseline_logreg_f1_score_test,
35                   best_logreg_f1_score_test,
36                   baseline_tree_f1_score_test,
37                   best_tree_f1_score_test,
38                   baseline_RF_f1_score_test,
39                   best_RF_f1_score_test,
40                   baseline_xgb_f1_score_test]
41
42 data = {'Model': models,
43         'Train Precision Score': train_precision_scores,
44         'Test Precision Score': test_precision_scores,
45         'Train F1 Score': train_f1_scores,
46         'Test F1 Score': test_f1_scores}
47
48 table = tabulate(data,
49                  headers='keys',
50                  tablefmt='presto')
51
52 print(table)
53

```

Model	Train Precision Score	Test Precision Score	Train F1 Score	Test F1 Score
Baseline Logreg	0.419432	0.432226	0.551109	0.561701
Best Logreg	0.430175	0.442249	0.558478	0.568822
Baseline Decision Tree	0.996498	0.699173	0.994441	0.715686
Best Decision Tree	1	1	0.801026	0.803713
Baseline Random Forest	0.994138	0.753302	0.994374	0.740895
Best Random Forest	1	1	0.801026	0.803713
Baseline XGBoost	0.93223	0.883058	0.836596	0.78499

▼ Tuned XGBoost

Hyperparameter Helper

n_estimators: This parameter specifies the number of boosting rounds (decision trees) to be built. The default value is 100.

learning_rate: Also known as the "eta" parameter, it controls the step size shrinkage during each boosting iteration. A lower learning rate requires more boosting rounds but can lead to better generalization. The default value is 0.1.

max_depth: This parameter sets the maximum depth of each decision tree in the boosting process. Higher values can make the model more complex and prone to overfitting. The default value is 3.

gamma: This parameter specifies the minimum loss reduction required to make a further partition on a leaf node during the tree-building process. Higher values lead to more conservative tree growth. The default value is 0.

min_child_weight: This parameter sets the minimum sum of instance weights (hessian) required in a child node. Higher values lead to more conservative tree growth. The default value is 1.

subsample: This parameter specifies the fraction of samples to be used for training each individual tree. Lower values make the model more conservative by using less data. The default value is 1 (use all samples).

colsample_bytree: This parameter specifies the fraction of features to be used for training each individual tree. Lower values make the model more conservative by using fewer features. The default value is 1 (use all features).

```

1 #tuning our XGB model
2 parameters = {
3     "estimator__n_estimators": [50, 100], #default 100
4     "estimator__learning_rate": [0.05, 0.1], #default 0.1
5     "estimator__max_depth": [3, 4], #default 3
6     "estimator__gamma": [0, 0.5], #default 0
7     "estimator__min_child_weight": [1, 3], #default 1
8     "estimator__subsample": [0.5, 1], #default 1
9     "estimator__colsample_bytree": [0.5, 1] #default 1
10 }
11
12 best_xgb = GridSearchCV(estimator=baseline_xgb,
13                         param_grid=parameters,
14                         scoring='precision',
15                         cv=3,
16                         n_jobs=-1
17 )
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
259
```

```

35 print(f" Test Recall score: {best_xgb_recall_score_test :.2%} ")
36 print(divider)
37
38 print(f" Train F1 score: {best_xgb_f1_score_train :.2%}")
39 print(f" Test F1 score: {best_xgb_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 84.85%
Test ROC Score: 84.79%
-----

```

```

-----
Train Precision score: 88.49%
Test Precision score: 89.55%
-----

```

```

-----
Train Recall score: 70.53%
Test Recall score: 70.38%
-----

```

```

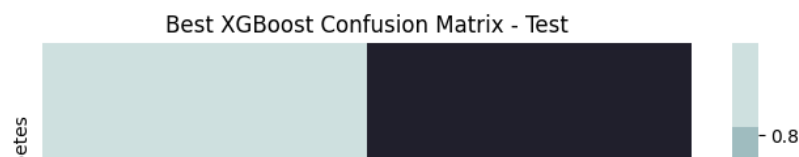
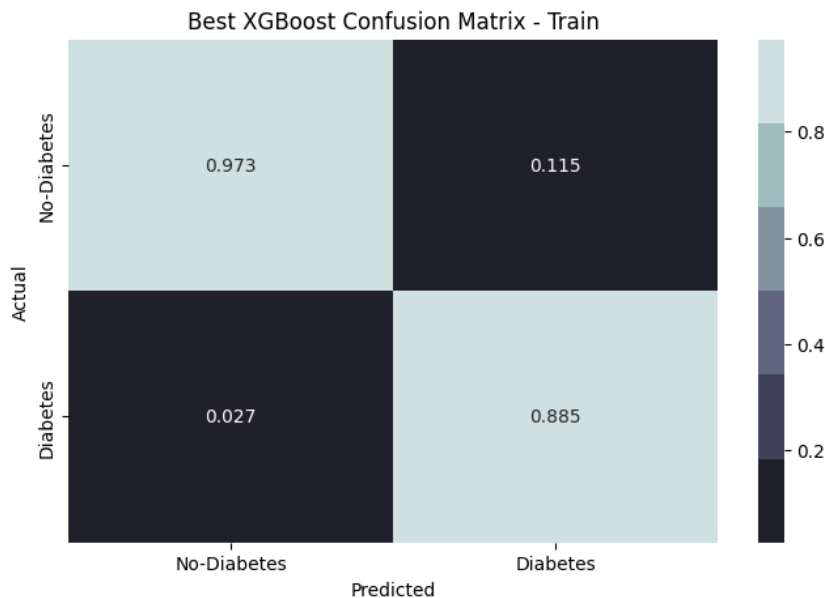
-----
Train F1 score: 78.50%
Test F1 score: 78.81%
-----

```

```

1 #compute the confusion matrix for the best xgboost - Train
2 train_pred = best_xgb.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the best xgboost - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pal)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Best XGBoost Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the best xgboost - Test
19 test_pred = best_xgb.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the best xgboost - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Best XGBoost Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



```

1 #run our comparison from all recent models to understand performance
2 models = ['Baseline Logreg',
3           'Best Logreg',
4           'Baseline Decision Tree',
5           'Best Decision Tree',
6           'Baseline Random Forest',
7           'Best Random Forest',
8           'Baseline XGBoost',
9           'Best XGBoost']
10
11 train_precision_scores = [baseline_logreg_precision_score_train,
12                            best_logreg_precision_score_train,
13                            baseline_tree_precision_score_train,
14                            best_tree_precision_score_train,
15                            baseline_RF_precision_score_train,
16                            best_RF_precision_score_train,
17                            baseline_xgb_precision_score_train,
18                            best_xgb_precision_score_train]
19
20 test_precision_scores = [baseline_logreg_precision_score_test,
21                           best_logreg_precision_score_test,
22                           baseline_tree_precision_score_test,
23                           best_tree_precision_score_test,
24                           baseline_RF_precision_score_test,
25                           best_RF_precision_score_test,
26                           baseline_xgb_precision_score_test,
27                           best_xgb_precision_score_test]
28
29 train_f1_scores = [baseline_logreg_f1_score_train,
30                    best_logreg_f1_score_train,
31                    baseline_tree_f1_score_train,
32                    best_tree_f1_score_train,
33                    baseline_RF_f1_score_train,
34                    best_RF_f1_score_train,
35                    baseline_xgb_f1_score_train,
36                    best_xgb_f1_score_train
37                    ]
38
39 test_f1_scores = [baseline_logreg_f1_score_test,
40                   best_logreg_f1_score_test,
41                   baseline_tree_f1_score_test,
42                   best_tree_f1_score_test,
43                   baseline_RF_f1_score_test,
44                   best_RF_f1_score_test,
45                   baseline_xgb_f1_score_test,
46                   best_xgb_f1_score_test]

```



```

47
48 #create a DataFrame with the data
49 data = {'Model': models,
50         'Train Precision Score': train_precision_scores,
51         'Test Precision Score': test_precision_scores,
52         'Train F1 Score': train_f1_scores,
53         'Test F1 Score': test_f1_scores}
54
55 df = pd.DataFrame(data)
56
57 #sort
58 df_sorted = df.sort_values(by='Test Precision Score', ascending=False)
59
60 #print
61 table = tabulate(df_sorted, headers='keys', tablefmt='presto')
62 print(table)
63

```

	Model	Train Precision Score	Test Precision Score	Train F1 Score	Test F1 Score
3	Best Decision Tree	1	1	0.801026	0.803713
5	Best Random Forest	1	1	0.801026	0.803713
7	Best XGBoost	0.884868	0.89547	0.784971	0.788142
6	Baseline XGBoost	0.93223	0.883058	0.836596	0.78499
4	Baseline Random Forest	0.994138	0.753302	0.994374	0.740895
2	Baseline Decision Tree	0.996498	0.699173	0.994441	0.715686
1	Best Logreg	0.430175	0.442249	0.558478	0.568822
0	Baseline Logreg	0.419432	0.432226	0.551109	0.561701

Our tuned xgboost model, although didn't perform much better on the F1 score, did perform better on the precision score between the baseline XGB and the tuned XGB models. Overall, still our best performing model is our Tuned or Best Random Forest model, predicting 100% of the positive cases, and scoring a .803713 on the f1 score. This means we didn't sacrifice much by the way of false positives in order to get our high precision score.

▼ Neural Network

▼ Baseline Neural Network

```

1 #moving along to our next model -- initialize the MLP classifier
2 baseline_mlp = ImPipeline(steps=[('sm', SMOTE(random_state=42)),
3                                  ('scale', StandardScaler()),
4                                  ('estimator', MLPClassifier(hidden_layer_sizes=(64, 32), activation='relu', solver='adam', r
5
6 #train model
7 baseline_mlp.fit(X_train, y_train);

```

/usr/local/lib/python3.10/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:686: ConvergenceWarning: Stochastic warnings.warn(

```

1 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
2 divider = ('-----' * 10)
3
4 #capture roc_auc for test, and train
5 baseline_mlp_roc_score_train = roc_auc_score(y_train, baseline_mlp.predict(X_train))
6 baseline_mlp_roc_score_test = roc_auc_score(y_test, baseline_mlp.predict(X_test))
7
8 #capture precision scores for test and train
9 baseline_mlp_precision_score_train_cv = cross_val_score(estimator=baseline_mlp, X=X_train, y=y_train,
10 # cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
11
12 baseline_mlp_precision_score_train = precision_score(y_train, baseline_mlp.predict(X_train))
13 baseline_mlp_precision_score_test = precision_score(y_test, baseline_mlp.predict(X_test))
14
15 #capture f1 scores for test and train
16 baseline_mlp_f1_score_train = f1_score(y_train, baseline_mlp.predict(X_train))
17 baseline_mlp_f1_score_test = f1_score(y_test, baseline_mlp.predict(X_test))
18
19 #capture recall scores for test and train
20 baseline_mlp_recall_score_train = recall_score(y_train, baseline_mlp.predict(X_train))
21 baseline_mlp_recall_score_test = recall_score(y_test, baseline_mlp.predict(X_test))
22

```

```

23 print('\n', "Performance Comparison", '\n')
24 print(divider)
25 print(f" Train ROC Score: {baseline_mlp_roc_score_train :.2%}")
26 print(f" Test ROC Score: {baseline_mlp_roc_score_test :.2%}")
27 print(divider)
28
29 print(f" Train Precision score: {baseline_mlp_precision_score_train :.2%}")
30 print(f" Test Precision score: {baseline_mlp_precision_score_test :.2%}")
31 #print(f" Mean Cross Validated Precision Score: {baseline_mlp_precision_score_train_cv :.2%}")
32 print(divider)
33
34 print(f" Train Recall score: {baseline_mlp_recall_score_train :.2%}")
35 print(f" Test Recall score: {baseline_mlp_recall_score_test :.2%}")
36 print(divider)
37
38 print(f" Train F1 score: {baseline_mlp_f1_score_train :.2%}")
39 print(f" Test F1 score: {baseline_mlp_f1_score_test :.2%}")
40 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 89.20%
Test ROC Score: 86.70%
-----

```

```

-----
Train Precision score: 54.68%
Test Precision score: 51.67%
-----

```

```

-----
Train Recall score: 84.86%
Test Recall score: 80.65%
-----

```

```

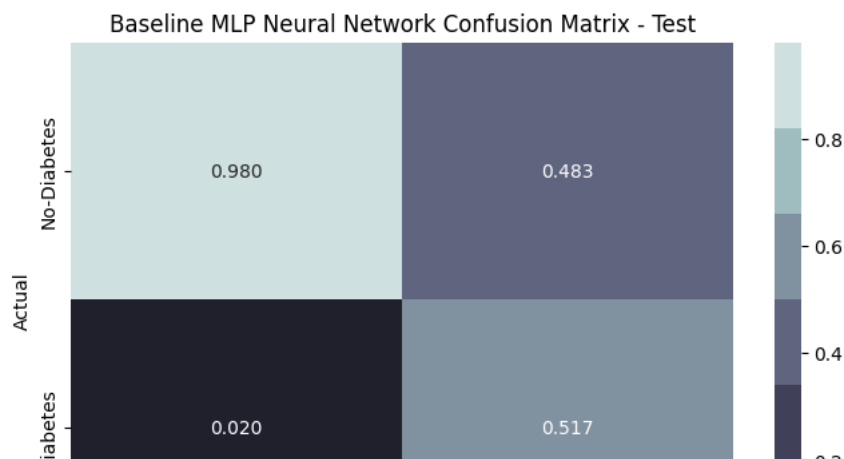
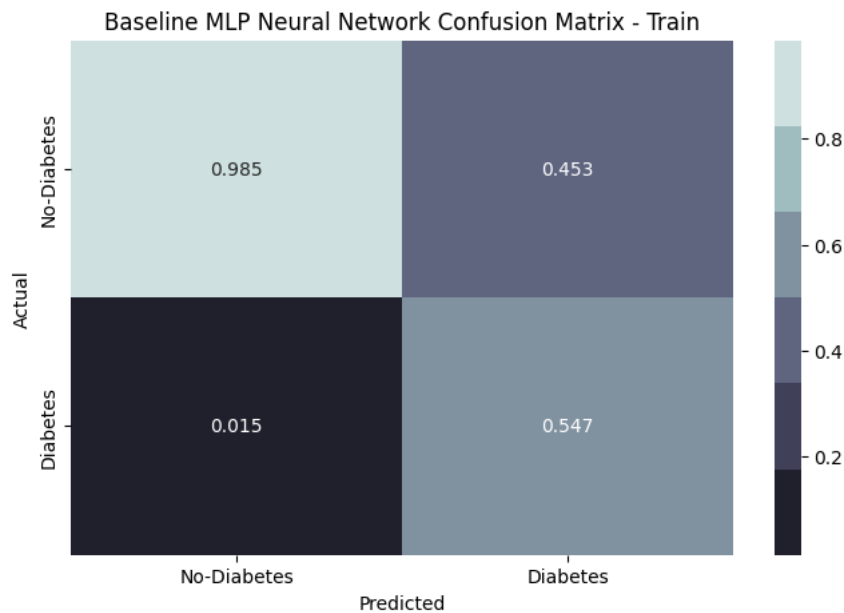
-----
Train F1 score: 66.51%
Test F1 score: 62.98%
-----

```

```

1 #compute the confusion matrix for the baseline mlp - Train
2 train_pred = baseline_mlp.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the baseline mlp - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pal)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Baseline MLP Neural Network Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the baseline mlp - Test
19 test_pred = baseline_mlp.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the baseline mlp - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Baseline MLP Neural Network Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()

```



```

1 #run our comparison from all recent models to understand performance
2 models = ['Baseline Logreg',
3           'Best Logreg',
4           'Baseline Decision Tree',
5           'Best Decision Tree',
6           'Baseline Random Forest',
7           'Best Random Forest',
8           'Baseline XGBoost',
9           'Best XGBoost',
10          'Baseline MLP Neural Network']
11
12 train_precision_scores = [baseline_logreg_precision_score_train,
13                           best_logreg_precision_score_train,
14                           baseline_tree_precision_score_train,
15                           best_tree_precision_score_train,
16                           baseline_RF_precision_score_train,
17                           best_RF_precision_score_train,
18                           baseline_xgb_precision_score_train,
19                           best_xgb_precision_score_train,
20                           baseline_mlp_precision_score_train]
21
22 test_precision_scores = [baseline_logreg_precision_score_test,
23                          best_logreg_precision_score_test,
24                          baseline_tree_precision_score_test,
25                          best_tree_precision_score_test,
26                          baseline_RF_precision_score_test,
27                          best_RF_precision_score_test,
28                          baseline_xgb_precision_score_test,
29                          best_xgb_precision_score_test,
30                          baseline_mlp_precision_score_test]
31
32 train_f1_scores = [baseline_logreg_f1_score_train,
33                   best_logreg_f1_score_train,

```

```

34         baseline_tree_f1_score_train,
35         best_tree_f1_score_train,
36         baseline_RF_f1_score_train,
37         best_RF_f1_score_train,
38         baseline_xgb_f1_score_train,
39         best_xgb_f1_score_train,
40         baseline_mlp_f1_score_train]
41
42 test_f1_scores = [baseline_logreg_f1_score_test,
43                  best_logreg_f1_score_test,
44                  baseline_tree_f1_score_test,
45                  best_tree_f1_score_test,
46                  baseline_RF_f1_score_test,
47                  best_RF_f1_score_test,
48                  baseline_xgb_f1_score_test,
49                  best_xgb_f1_score_test,
50                  baseline_mlp_f1_score_test]
51
52 data = {'Model': models,
53         'Train Precision Score': train_precision_scores,
54         'Test Precision Score': test_precision_scores,
55         'Train F1 Score': train_f1_scores,
56         'Test F1 Score': test_f1_scores}
57
58 table = tabulate(data,
59                 headers='keys',
60                 tablefmt='presto')
61
62 print(table)
63

```

Model	Train Precision Score	Test Precision Score	Train F1 Score	Test F1 Score
Baseline Logreg	0.419432	0.432226	0.551109	0.561701
Best Logreg	0.430175	0.442249	0.558478	0.568822
Baseline Decision Tree	0.996498	0.699173	0.994441	0.715686
Best Decision Tree	1	1	0.801026	0.803713
Baseline Random Forest	0.994138	0.753302	0.994374	0.740895
Best Random Forest	1	1	0.801026	0.803713
Baseline XGBoost	0.93223	0.883058	0.836596	0.78499
Best XGBoost	0.884868	0.89547	0.784971	0.788142
Baseline MLP Neural Network	0.546829	0.516667	0.665093	0.629834

This performance isn't surprising based on the error message above. Needless to say, this model needs tuned to really evaluate the performance of this classifier.

▼ Tuned Neural Network

Hyperparameter Helper

hidden_layer_sizes: This parameter defines the architecture of the neural network by specifying the number of neurons in each hidden layer. The values you provided indicate different configurations, such as a single hidden layer with 50 neurons, a single hidden layer with 100 neurons, two hidden layers each with 50 neurons, and two hidden layers each with 100 neurons.

activation: This parameter determines the activation function used in the neural network. The values 'relu' and 'tanh' correspond to the rectified linear unit and hyperbolic tangent activation functions, respectively. Different activation functions can have different effects on the learning process and performance of the neural network.

learning_rate_init: This parameter sets the initial learning rate for the neural network. The learning rate controls the step size during gradient descent, affecting how quickly the network learns. The values you provided are different options for the initial learning rate.

alpha: This parameter represents the L2 regularization term in the neural network's cost function. It helps prevent overfitting by adding a penalty term to the loss function. The values you provided are different options for the regularization strength.

```

1 #tuning hyperparameters for our gridsearch
2 mlpparameters = {
3     'estimator__hidden_layer_sizes': [(50,), (100,), (50, 50), (100, 100)],
4     'estimator__activation': ['relu', 'tanh'],
5     'estimator__learning_rate_init': [0.001, 0.01, 0.1],
6     'estimator__alpha': [0.0001, 0.001, 0.01]
7 }
8
9 best_mlp = GridSearchCV(estimator=baseline_mlp,

```

```

10         param_grid=mlpparameters,
11         scoring='precision',
12         cv=3,
13         n_jobs=-1
14     )
15
16
17 #train the pipeline based on our most appropriate parameters
18 best_mlp.fit(X_train,
19             y_train)
20
21 best_mlp.best_params_
22
23 {'estimator__activation': 'relu',
24  'estimator__alpha': 0.0001,
25  'estimator__hidden_layer_sizes': (100, 100),
26  'estimator__learning_rate_init': 0.01}
27
28
29 #scoring print out adapted from others -- Eva Mizer, and Aysu Erdemir.
30 divider = ('-----' * 10)
31
32 #capture roc_auc for test, and train
33 best_mlp_roc_score_train = roc_auc_score(y_train, best_mlp.predict(X_train))
34 best_mlp_roc_score_test = roc_auc_score(y_test, best_mlp.predict(X_test))
35
36 #capture precision scores for test and train
37 #best_mlp_precision_score_train_cv = cross_val_score(estimator=best_mlp, X=X_train, y=y_train,
38 #                                                    cv=StratifiedKFold(shuffle=True), scoring='precision').mean()
39
40 best_mlp_precision_score_train = precision_score(y_train, best_mlp.predict(X_train))
41 best_mlp_precision_score_test = precision_score(y_test, best_mlp.predict(X_test))
42
43 #capture f1 scores for test and train
44 best_mlp_f1_score_train = f1_score(y_train, best_mlp.predict(X_train))
45 best_mlp_f1_score_test = f1_score(y_test, best_mlp.predict(X_test))
46
47 #capture recall scores for test and train
48 best_mlp_recall_score_train = recall_score(y_train, best_mlp.predict(X_train))
49 best_mlp_recall_score_test = recall_score(y_test, best_mlp.predict(X_test))
50
51 print('\n', "Performance Comparison", '\n')
52 print(divider)
53 print(f" Train ROC Score: {best_mlp_roc_score_train :.2%}")
54 print(f" Test ROC Score: {best_mlp_roc_score_test :.2%}")
55 print(divider)
56
57 print(f" Train Precision score: {best_mlp_precision_score_train :.2%}")
58 print(f" Test Precision score: {best_mlp_precision_score_test :.2%}")
59 #print(f" Mean Cross Validated Precision Score: {best_mlp_precision_score_train_cv :.2%}")
60 print(divider)
61
62 print(f" Train Recall score: {best_mlp_recall_score_train :.2%}")
63 print(f" Test Recall score: {best_mlp_recall_score_test :.2%}")
64 print(divider)
65
66 print(f" Train F1 score: {best_mlp_f1_score_train :.2%}")
67 print(f" Test F1 score: {best_mlp_f1_score_test :.2%}")
68 print(divider, '\n')

```

Performance Comparison

```

-----
Train ROC Score: 87.89%
Test ROC Score: 86.02%

```

```

-----
Train Precision score: 57.01%
Test Precision score: 54.32%

```

```

-----
Train Recall score: 81.42%
Test Recall score: 78.37%

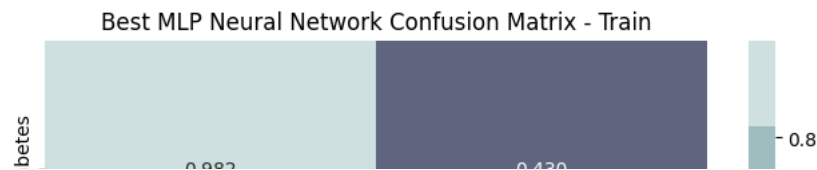
```

```

-----
Train F1 score: 67.06%
Test F1 score: 64.16%
-----

```

```
1 #compute the confusion matrix for the best mlp - Train
2 train_pred = best_mlp.predict(X_train)
3 train_cm = confusion_matrix(y_train, train_pred)
4
5 #normalize the confusion matrix by dividing each column by its sum
6 train_cm_normalized = train_cm / train_cm.sum(axis=0, keepdims=True)
7
8 #plotting the confusion matrix for the best mlp - Train
9 plt.figure(figsize=(8, 5))
10 sns.heatmap(train_cm_normalized, annot=True, fmt='.3f', cmap=pal)
11 plt.xlabel('Predicted')
12 plt.ylabel('Actual')
13 plt.title("Best MLP Neural Network Confusion Matrix - Train")
14 plt.xticks(ticks=[0.5, 1.5], labels=labels)
15 plt.yticks(ticks=[0.5, 1.5], labels=labels)
16 plt.show()
17
18 #compute the confusion matrix for the best mlp - Test
19 test_pred = best_mlp.predict(X_test)
20 test_cm = confusion_matrix(y_test, test_pred)
21
22 #normalize the confusion matrix by dividing each column by its sum
23 test_cm_normalized = test_cm / test_cm.sum(axis=0, keepdims=True)
24
25 #plotting the confusion matrix for the best mlp - Test
26 plt.figure(figsize=(8, 5))
27 sns.heatmap(test_cm_normalized, annot=True, fmt='.3f', cmap=pal)
28 plt.xlabel('Predicted')
29 plt.ylabel('Actual')
30 plt.title("Best MLP Neural Network Confusion Matrix - Test")
31 plt.xticks(ticks=[0.5, 1.5], labels=labels)
32 plt.yticks(ticks=[0.5, 1.5], labels=labels)
33 plt.show()
```



Overall our MLP approach proved to be better at recall, but worse at our primary metric of precision. Thus this brought down the overall f1 score, and put the algorithm towards the middle of our list. We will move forward with our XGBoost algorithm and further summarize our results/ comparisons below.



Results



```
1 #run our comparison from all recent models to understand performance
2 models = ['Baseline Logreg',
3           'Best Logreg',
4           'Baseline Decision Tree',
5           'Best Decision Tree',
6           'Baseline Random Forest',
7           'Best Random Forest',
8           'Baseline XGBoost',
9           'Best XGBoost',
10          'Baseline MLP Neural Network',
11          'Best MLP Neural Network'
12          ]
13
14 train_precision_scores = [baseline_logreg_precision_score_train,
15                           best_logreg_precision_score_train,
16                           baseline_tree_precision_score_train,
17                           best_tree_precision_score_train,
18                           baseline_RF_precision_score_train,
19                           best_RF_precision_score_train,
20                           baseline_xgb_precision_score_train,
21                           best_xgb_precision_score_train,
22                           baseline_mlp_precision_score_train,
23                           best_mlp_precision_score_train
24                           ]
25
26 test_precision_scores = [baseline_logreg_precision_score_test,
27                          best_logreg_precision_score_test,
28                          baseline_tree_precision_score_test,
29                          best_tree_precision_score_test,
30                          baseline_RF_precision_score_test,
31                          best_RF_precision_score_test,
32                          baseline_xgb_precision_score_test,
33                          best_xgb_precision_score_test,
34                          baseline_mlp_precision_score_test,
35                          best_mlp_precision_score_test
36                          ]
37
38 train_f1_scores = [baseline_logreg_f1_score_train,
39                   best_logreg_f1_score_train,
40                   baseline_tree_f1_score_train,
41                   best_tree_f1_score_train,
42                   baseline_RF_f1_score_train,
43                   best_RF_f1_score_train,
44                   baseline_xgb_f1_score_train,
45                   best_xgb_f1_score_train,
46                   baseline_mlp_f1_score_train,
47                   best_mlp_f1_score_train
48                   ]
49
50 test_f1_scores = [baseline_logreg_f1_score_test,
51                  best_logreg_f1_score_test,
52                  baseline_tree_f1_score_test,
53                  best_tree_f1_score_test,
54                  baseline_RF_f1_score_test,
55                  best_RF_f1_score_test,
56                  baseline_xgb_f1_score_test,
57                  best_xgb_f1_score_test,
58                  baseline_mlp_f1_score_test,
59                  best_mlp_f1_score_test
```

```

60         ]
61
62 train_recall_scores = [baseline_logreg_recall_score_train,
63                         best_logreg_recall_score_train,
64                         baseline_tree_recall_score_train,
65                         best_tree_recall_score_train,
66                         baseline_RF_recall_score_train,
67                         best_RF_recall_score_train,
68                         baseline_xgb_recall_score_train,
69                         best_xgb_recall_score_train,
70                         baseline_mlp_recall_score_train,
71                         best_mlp_recall_score_train
72                     ]
73
74 test_recall_scores = [baseline_logreg_recall_score_test,
75                       best_logreg_recall_score_test,
76                       baseline_tree_recall_score_test,
77                       best_tree_recall_score_test,
78                       baseline_RF_recall_score_test,
79                       best_RF_recall_score_test,
80                       baseline_xgb_recall_score_test,
81                       best_xgb_recall_score_test,
82                       baseline_mlp_recall_score_test,
83                       best_mlp_recall_score_test
84                   ]
85
86 data = {'Model': models,
87         'Train Precision Score': train_precision_scores,
88         'Test Precision Score': test_precision_scores,
89         'Train F1 Score': train_f1_scores,
90         'Test F1 Score': test_f1_scores,
91         'Train Recall Score': train_recall_scores,
92         'Test Recall Score': test_recall_scores}
93
94 table = tabulate(data,
95                 headers='keys',
96                 tablefmt='presto')
97
98 print(table)
99

```

Model	Train Precision Score	Test Precision Score	Train F1 Score	Test F1 Score	Tra
Baseline Logreg	0.419432	0.432226	0.551109	0.561701	
Best Logreg	0.430175	0.442249	0.558478	0.568822	
Baseline Decision Tree	0.996498	0.699173	0.994441	0.715686	
Best Decision Tree	1	1	0.801026	0.803713	
Baseline Random Forest	0.994138	0.753302	0.994374	0.740895	
Best Random Forest	1	1	0.801026	0.803713	
Baseline XGBoost	0.93223	0.883058	0.836596	0.78499	
Best XGBoost	0.884868	0.89547	0.784971	0.788142	
Baseline MLP Neural Network	0.546829	0.516667	0.665093	0.629834	
Best MLP Neural Network	0.570081	0.543183	0.670627	0.641629	

```

1 #creating a visual of the table above for our readme and presentation
2 train_precision_scores = [baseline_logreg_precision_score_train,
3                           best_logreg_precision_score_train,
4                           baseline_tree_precision_score_train,
5                           best_tree_precision_score_train,
6                           baseline_RF_precision_score_train,
7                           best_RF_precision_score_train,
8                           baseline_xgb_precision_score_train,
9                           best_xgb_precision_score_train,
10                          baseline_mlp_precision_score_train,
11                          best_mlp_precision_score_train
12                      ]
13
14 test_precision_scores = [baseline_logreg_precision_score_test,
15                          best_logreg_precision_score_test,
16                          baseline_tree_precision_score_test,
17                          best_tree_precision_score_test,
18                          baseline_RF_precision_score_test,
19                          best_RF_precision_score_test,
20                          baseline_xgb_precision_score_test,
21                          best_xgb_precision_score_test,
22                          baseline_mlp_precision_score_test,
23                          best_mlp_precision_score_test

```



```

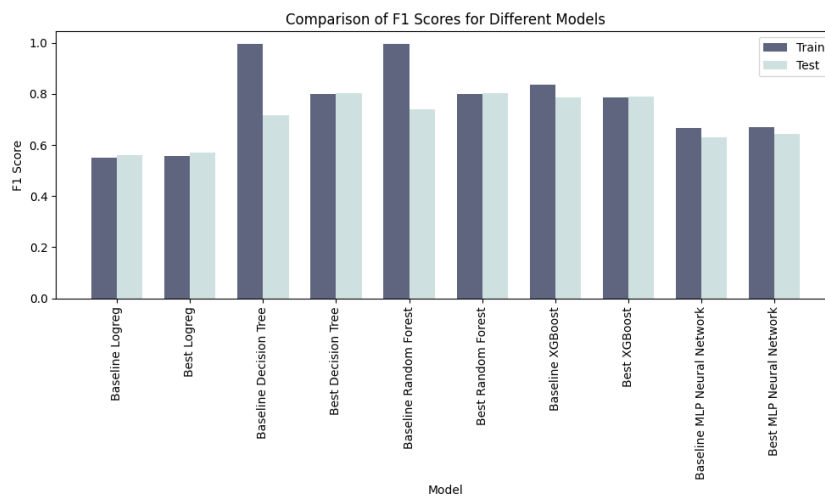
24         ]
25
26 train_f1_scores = [baseline_logreg_f1_score_train,
27                    best_logreg_f1_score_train,
28                    baseline_tree_f1_score_train,
29                    best_tree_f1_score_train,
30                    baseline_RF_f1_score_train,
31                    best_RF_f1_score_train,
32                    baseline_xgb_f1_score_train,
33                    best_xgb_f1_score_train,
34                    baseline_mlp_f1_score_train,
35                    best_mlp_f1_score_train
36                ]
37
38 test_f1_scores = [baseline_logreg_f1_score_test,
39                  best_logreg_f1_score_test,
40                  baseline_tree_f1_score_test,
41                  best_tree_f1_score_test,
42                  baseline_RF_f1_score_test,
43                  best_RF_f1_score_test,
44                  baseline_xgb_f1_score_test,
45                  best_xgb_f1_score_test,
46                  baseline_mlp_f1_score_test,
47                  best_mlp_f1_score_test
48                ]
49
50 #sort
51 sorted_indices = np.argsort(test_precision_scores[::-1])
52 models_sorted = [models[i] for i in sorted_indices]
53 train_precision_scores_sorted = [train_precision_scores[i] for i in sorted_indices]
54 test_precision_scores_sorted = [test_precision_scores[i] for i in sorted_indices]
55
56 #set the positions of the bars on the x-axis
57 r1 = np.arange(len(models_sorted))
58 r2 = [x + bar_width for x in r1]
59
60 #create the bar chart
61 plt.figure(figsize=(10, 6))
62 plt.bar(r1, train_precision_scores_sorted, color=pal[2], width=bar_width, label='Train')
63 plt.bar(r2, test_precision_scores_sorted, color=pal[5], width=bar_width, label='Test')
64 plt.xticks([r + bar_width/2 for r in r1], models_sorted, rotation='vertical')
65 plt.xlabel('Model')
66 plt.ylabel('Precision Score')
67 plt.title('Comparison of Precision Scores for Different Models')
68 plt.legend()
69
70 plt.tight_layout()
71 plt.show()

```

```

1 #creating a second version looking at f1 scores
2 #set the width of the bars
3 bar_width = 0.35
4
5 #set the positions of the bars on the x-axis
6 r1 = np.arange(len(models))
7 r2 = [x + bar_width for x in r1]
8
9 #create the bar chart for f1 scores
10 plt.figure(figsize=(10, 6))
11 plt.bar(r1, train_f1_scores, color=pal[2], width=bar_width, label='Train')
12 plt.bar(r2, test_f1_scores, color=pal[5], width=bar_width, label='Test')
13 plt.xticks([r + bar_width/2 for r in r1], models, rotation='vertical')
14 plt.xlabel('Model')
15 plt.ylabel('F1 Score')
16 plt.title('Comparison of F1 Scores for Different Models')
17 plt.legend()
18
19 #display the chart
20 plt.tight_layout()
21 plt.show()

```



** Need insight here

▼ Feature Importance

```

1 #taking a look at our feature importances
2 best_RF.estimator.steps[2][1].feature_importances_

array([0.17051672, 0.00343996, 0.00206454, 0.10549377, 0.41114197,
        0.25444642, 0.00724638, 0.0100285 , 0.00589655, 0.00741992,
        0.01346433, 0.00884094])

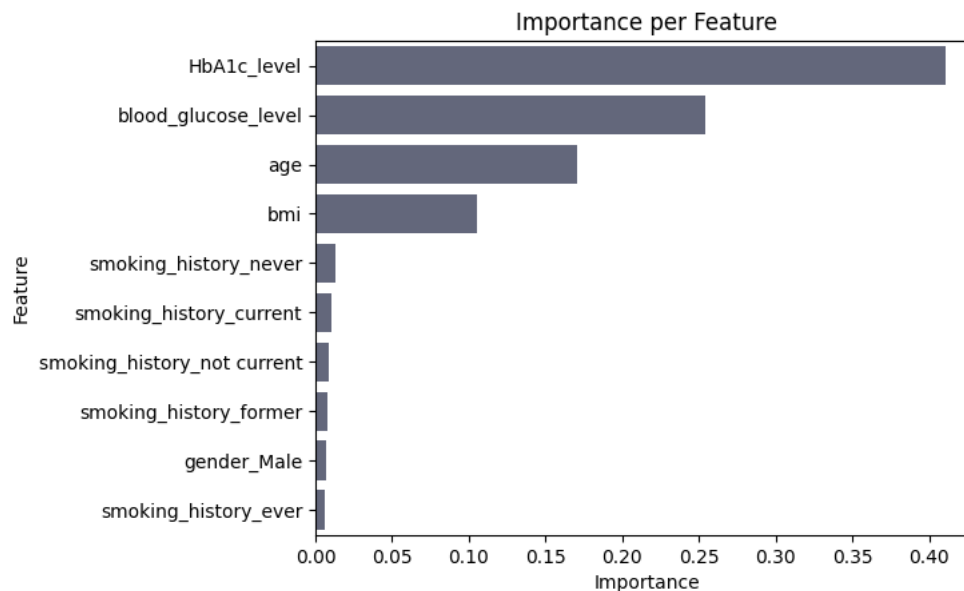
1 #columns on our dataframe
2 df_X_train_copy.columns

Index(['age', 'hypertension', 'heart_disease', 'bmi', 'HbA1c_level',
       'blood_glucose_level', 'gender_Male', 'smoking_history_current',

```

```
'smoking_history_ever', 'smoking_history_former',
'smoking_history_never', 'smoking_history_not current'],
dtype='object')
```

```
1 #visualize our Feature importances
2
3 feature_imp_rf = pd.DataFrame(pd.Series(best_RF.estimator.steps[2][1].feature_importances_, index=df_X_train_copy.columns).sc
4 feature_imp_rf = feature_imp_rf.head(10)
5 feature_imp_rf['index'] = feature_imp_rf.index
6 feature_imp_rf.head()
7
8 ax = sns.barplot(x=feature_imp_rf[0], y=feature_imp_rf['index'], data=feature_imp_rf, color=pal[2])
9 ax.set_xlabel('Importance')
10 ax.set_ylabel('Feature')
11 ax.set_title('Importance per Feature');
```



Conclusion

In our diabetes classification problem, we aimed to develop models that could accurately predict the presence of diabetes based on various features. We evaluated the performance of several models, including logistic regression, decision trees, random forest, XGBoost, and MLP neural network.

Overall, our best-performing model for our metrics of interest was the tuned XGBoost, achieving a precision score of 88% and 89% on the train and test sets respectively. It demonstrated excellent accuracy in identifying true positive cases of diabetes while minimizing false positives.

When considering the F1 score, which balances precision and recall, the best decision tree and random forest models showed the highest scores. These however were being influenced by the extremely high precision score, not taking into account the recall performance (as much). When looking at our XGBoost models, these models achieved F1 scores of around 80% on the test set, indicating a good balance between precision and recall. They were not the highest, but they also had performed better and more evenly across all metrics.

In summary, our models demonstrated strong performance in accurately classifying diabetes cases. The XGBoost, with its balanced performance across precision, F1 score, and ultimately recall show promising potential for accurately predicting diabetes in future applications.

Now that we have a well trained model, we can use it to make predictions on new data. We were also able to gain insight into what features were the most impactful in the diagnosis of diabetes. Moving forward we can suggest the following steps to optimize our interventions:

1. Run the algorithm on new data.
2. Continually evolve the datasets that are being used for prediction.
3. Try to understand time, and impact of additional metrics in #2 and early diagnosis.
4. Evaluate impact of interventions on classified population vs those that were not classified for programming.
5. Load data into centralized repository for sharing into operational systems.