1. Re-implementation of the article

The article from [2] is about proposing Machine Learning solution to classify type 2 diabetes disease patients from a group of females at least 21 years old of Pima Indian heritage.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

1.1. Data preprocessing

The dataset consists of 8 features and 1 output. The problem is a binary classification. The first step in the study said to preprocess the data. They said that the zero values in any of the feature columns such as pregnancy, blood pressure, skin thickness, insulin and BMI are considered as missing values because zero values are not biologically conceivable.

```
# Replace all value that is 0 with NAN
for column in df.columns:
    if column != 'Outcome':
        df[column].replace(0, np.nan, inplace=True)
```

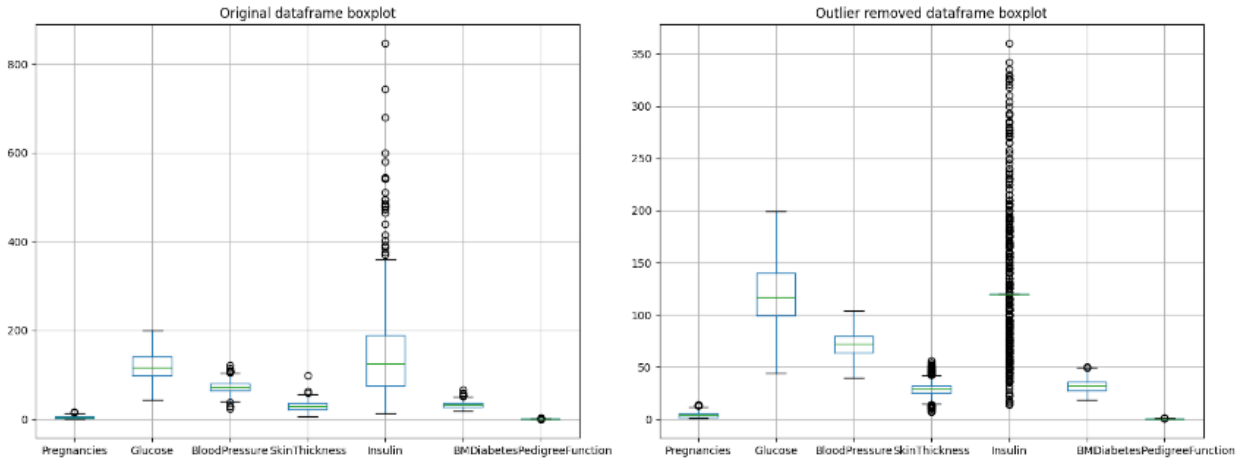
After replacing all value that is 0 with NAN, the paper said to detect outliers with both z-score and IQR, then replace the outliers with NAN values.

```
def detect_outliers_z_iqr(df: pd.DataFrame, columns: list[str]) -> pd.DataFrame:
    """
    Detect outliers using both z-score and iqr
    - df (pd.DataFrame): input df
    - columns (list[str]): columns selected

    Returns:
    - pd.DataFrame: applied outlier detection df
    """
    df_copy = df.copy()
    for column in columns:
        # Z-score method
        z_scores = np.abs(stats.zscore(df_copy[column]))
        z_outliers = z_scores > 3

        # IQR method
        Q1 = df_copy[column].quantile(0.25)
        Q3 = df_copy[column].quantile(0.75)
        IQR = Q3 - Q1
        iqr_outliers = (df_copy[column] < (Q1 - 1.5 * IQR)) | (df_copy[column] > (Q3 + 1.5 * IQR))

        # Combine both method, true if outlier is in either method
        outliers = z_outliers | iqr_outliers

        # Print debug info
        print(f"Column: {column}")
        print(f"Z-score outliers: {df_copy[column][z_outliers].values}")
        print(f"IQR outliers: {df_copy[column][iqr_outliers].values}")

        # Replace outlier with nan
        df_copy.loc[outliers, column] = np.nan

    return df_copy
```

```
columns = ['Pregnancies','Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction']
df2 = detect_outliers_z_iqr(df, columns)
```

```
Column: Pregnancies
Z-score outliers: []
IQR outliers: [15. 17.]
Column: Glucose
Z-score outliers: []
IQR outliers: []
Column: BloodPressure
Z-score outliers: []
IQR outliers: [ 30. 110. 108. 122.  30. 110. 108. 110.  24.  38. 106. 106. 106. 114.]
Column: SkinThickness
Z-score outliers: []
IQR outliers: [60. 63. 99.]
Column: Insulin
Z-score outliers: []
IQR outliers: [543. 846. 495. 485. 495. 478. 744. 370. 680. 402. 375. 545. 465. 415.
 579. 474. 480. 600. 440. 540. 480. 387. 392. 510.]
Column: BMI
Z-score outliers: []
IQR outliers: [53.2 55.  67.1 52.3 52.3 52.9 59.4 57.3]
Column: DiabetesPedigreeFunction
Z-score outliers: [2.288 1.893 1.781 2.329 1.476 2.137 1.731 1.6   2.42  1.699 1.698]
IQR outliers: [2.288 1.441 1.39  1.893 1.781 1.222 1.4   1.321 1.224 2.329 1.318 1.213
 1.353 1.224 1.391 1.476 2.137 1.731 1.268 1.6   2.42  1.251 1.699 1.258
 1.282 1.698 1.461 1.292 1.394]
```

Now, we need to impute those missing values with the median. Median imputation is a simple and robust imputer against skewed data. Here is a box plots before and after removing outliers.




```
df2.Insulin.describe()
```

```
count    768.000000
mean     126.075523
std       51.909477
min       14.000000
25%      120.000000
50%      120.000000
75%      120.000000
max      360.000000
Name: Insulin, dtype: float64
```

On the outlier removed box plot, the Insulin feature seems to be highly concentrated around the value 120, and there are extreme values that pull the whiskers and make the plot looks compressed. I will just leave it as it is for now. The final part of the data cleaning is to standardize the data and oversample the imbalance label using SMOTE.

1.2. ML based models

They have proposed an ML and DL approach, but since this task only asks for ML, I will be focusing on that. They have 4 different classic Machine Learning models, which are Decision Tree (DT), SVM, Random Forest (RF) and Stacking Ensemble. The Stacking Ensemble is an ensemble method where they use DT, RF, SVM as base models. These models each output their own prediction probabilities, then those predictions will be fed into a meta learner (Logistic Regression in this case) to derive at the final output. The table below will show the outcome of their models.

Protocol	Algorithm	Accuracy	Precision	Recall	F1-Score
Train-Test	DT	65.08	.65	.65	.65
	RF	79.33	.80	.79	.79
	SVM	69.03	.69	.69	.69
	Staking Ensemble	75.03	.75	.75	.75
	DNN1	68.80	.64	.74	.69
	DNN2	64.15	.64	.75	.68
	DNN3	65.40	.68	.69	.67
	Staking Ensemble	68.05	.68	.68	.62
Cross-validation	DT	68.31	.65	.68	.67
	RF	76.81	.77	.79	.78
	SVM	68.61	.68	.70	.69
	Staking Ensemble	77.10	.68	.70	.69
	DNN1	63.50	.63	.66	.64
	DNN2	64.50	.64	.65	.65
	DNN3	63.50	.64	.62	.63
	Staking Ensemble	65.50	.66	.65	.65

I suspect that the “Train-Test” is the stand-alone models, and the “Cross-validation” is when they find the most optimal hyperparameters for the model using grid search and cross-validation. I will show each model the “Train-Test” and “Cross-validation” predictions. Using the same 70-30 split as the study, here are my outcomes.

First, its Random Forest. Here is the performance of “Train-Test” on RF.

```python
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train, y_train)

# y pred
y_pred = rf.predict(X_test)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Classification report
report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Test Accuracy: 78.33%
Precision: 78.54%
Recall: 78.33%
F1-Score: 78.29%
```

My stand alone RF performance has similar accuracy to the paper's stand alone RF, only 1% difference.

Here is the “Cross-validation” for RF.

Best parameters: {'split_criterion': 1, 'n_estimators': 250, 'min_samples_split': 8, 'min_samples_leaf': 1, 'max_depth': 25}

```python
# Best hyperparameter model
rf = RandomForestClassifier(
    max_depth = 25,
    min_samples_leaf = 1,
    min_samples_split = 8,
    n_estimators = 250,
    split_criterion = 1,
    random_state = 42
)
cv_scores = cross_val_score(rf, X_train, y_train, cv=5, scoring='accuracy')
print(f"Cross-validation accuracy: {cv_scores.mean():.4f} (± {cv_scores.std():.4f})")

rf.fit(X_train, y_train)

# y pred
y_pred = rf.predict(X_test)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Classification report
report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Cross-validation accuracy: 0.8200 (± 0.0131)
Test Accuracy: 78.33%
Precision: 78.69%
Recall: 78.33%
F1-Score: 78.26%
```

The cross validation and the default RF model are not much different. My cv model is a bit higher than the study, I will need the predictions of more models to conclude that this is just the result of randomness or there might be some differences in our preprocessing steps.

The next model is DT "Train-Test" and "Cross-validation". Here is the DT "Train-Test".

```
clf=DecisionTreeClassifier(random_state=0)
clf.fit(X_train,y_train)

y_pred=clf.predict(X_test)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Classification report
report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Test Accuracy: 71.33%
Precision: 71.50%
Recall: 71.33%
F1-Score: 71.26%
```

My stand-alone DT is 6% higher than in the study, which is a decent amout, and after running the model twice with different random state, I can confirm that the predition differences is not due to randomness but to different preprocessing step. I have read the dataset description and preprocessing (3.1 and 3.2) and implement based on the text, but there are still a decent different in the accuracy. Perhaps, in the outlier removing step, the boundary that they consider which data is outlier is different. This might be the most logical explanation I can think of to explain the differences. Here is the "Cross-validation" for DT.

```
grid_param={"criterion":["gini","entropy"],
            "splitter":["best","random"],
            "max_depth":np.arange(10, 50, 5),
            "min_samples_leaf":range(1,20,1),
            "min_samples_split":range(2,20,1)
           }
grid_search=GridSearchCV(estimator=clf,param_grid=grid_param,cv=5,n_jobs=-1)
grid_search.fit(X_train,y_train)
print(grid_search.best_params_)
```

```
clf=DecisionTreeClassifier(criterion = 'entropy', max_depth = 10, min_samples_leaf = 19, min_samples_split = 2, splitt
clf.fit(X_train,y_train)
y_predicted=clf.predict(X_test)

# Classification report
accuracy = accuracy_score(y_test,y_predicted) * 100
print(f"Accuracy: {accuracy:.2f}%")

report = classification_report(y_test, y_predicted, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Accuracy: 71.67%
Precision: 71.67%
Recall: 71.67%
F1-Score: 71.66%
```

My “Cross-validation” for DT is not much different from my “Train-Test”, but it is still higher than the study.

Here are the “Train-Test” and “Cross-validation” for SVM.

Train set

```
svm = SVC()
svm.fit(X_train, y_train)

# Classification report
y_pred = svm.predict(X_test)

accuracy = accuracy_score(y_test,y_pred) * 100
print(f"Accuracy: {accuracy:.2f}%")

report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Accuracy: 79.33%
Precision: 79.66%
Recall: 79.33%
F1-Score: 79.27%
```

Cross validation

```
# SVC model
svm = SVC()

# Grid search
param_grid = {
    'C': np.logspace(-3, 2, 10),
    'kernel': ['rbf'],
    'gamma': ['scale']
}
grid_search = GridSearchCV(svm, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Best hyperparameter
best_params = grid_search.best_params_
print(f"Best parameters: {best_params}")

# Model accuracy
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)
accuracy = np.mean(y_pred == y_test)
print(f"Test Accuracy: {accuracy}")
```

```
Best parameters: {'C': 2.1544346900318843, 'gamma': 'scale', 'kernel': 'rbf'}
Test Accuracy: 0.8
```

```
# Classification report
y_pred = best_model.predict(X_test)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Test Accuracy: 80.00%
Precision: 80.25%
Recall: 80.00%
F1-Score: 79.95%
```

Similar to DT, my SVM (stand alone and cross validated) does a better job at classifiying than the research.

After the base models have been trained, I have chosen the best model either “Train-Test” or “Cross-validation” as estimators for the Stacking Classifier. The meta learner is Logistic Regression. The predictions for both “Train-Test” and “Cross-validation” are exactly the same.

```
stacking_clf = StackingClassifier(
    estimators=[
        ('dt', DecisionTreeClassifier(random_state=42, criterion='gini', max_depth=10,
                                      min_samples_leaf=6, min_samples_split=15, splitter='best')),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, C=2.1544346900318843,
                    gamma='scale', kernel='rbf'))
    ],
    final_estimator=LogisticRegression(random_state=42),
    cv=5
)
stacking_clf.fit(X_train, y_train)

# Classification report
y_pred = stacking_clf.predict(X_test)

accuracy = accuracy_score(y_test,y_pred) * 100
print(f"Accuracy: {accuracy:.2f}%")

report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
[I] [13:42:56.555909] Unused keyword parameter: random_state during cuML estimator initialization
Accuracy: 78.00%
Precision: 78.39%
Recall: 78.00%
F1-Score: 77.91%
```

My stacking classifier does better prediction than the paper. The difference is not much, but then again, this might be due to the combination of randomness and differences in the boundary of the outlier removal.

2. My ML solution

2.1. Data Preprocessing

My data cleaning step is similar to the study, but I have made some changes. The first change is in the zero values. [2] said that "the zero values found in qualities like pregnancy, blood pressure, skin thickness, insulin, and BMI are not biologically conceivable", while I agree with most of it, I think that pregnancy or "Pregnancies" can be zero. Zero value in pregnancy just means that the person has not been impregnated. For that, I do not replace columns "Pregnancies" and "Outcome" zero values with NAN.

```
for column in df.columns:
    if column not in ['Outcome', 'Pregnancies']:
        df[column].replace(0, np.nan, inplace=True)
```

Then, I follow the same outlier removal technique of using z-score and IQR and impute the outliers and zero values with the median.

```
columns = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age']
df2 = detect_outliers_z_iqr(df, columns)
```

Clip `Insulin` to a reasonable range since I don't like extreme values.

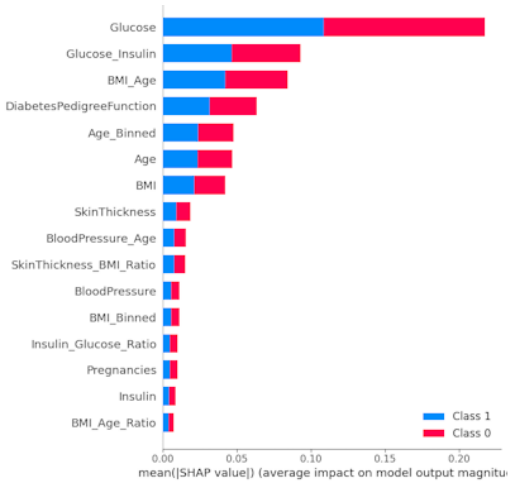
```
df2['Insulin'] = df['Insulin'].clip(upper=250.0)
```

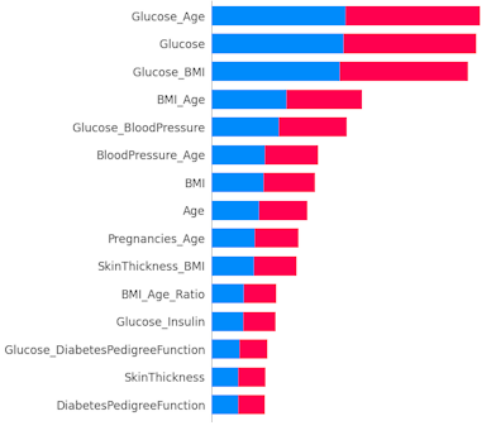
```
# Replace NaN, inf, 0, and 0.0 values with the median for each column
for column in columns:
    median_value = df2[column].median()
    df2[column].replace([np.nan, np.inf, -np.inf], median_value, inplace=True)
```

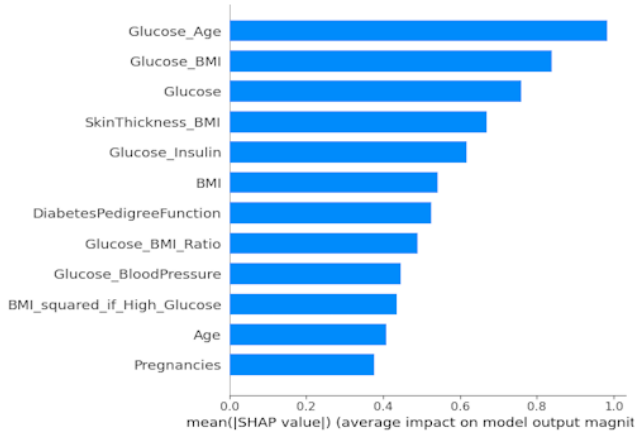
One thing that is different in this case is that I clip the "Insulin" to limit at 250. Like I said earlier, the majority of "Insulin" data is concentrated around 120, so extreme values compressed the box plot. I fear this will interfere with the model prediction, so I clip it.

2.2. Feature Engineer

Feature Engineer is creating new features from existing ones. This step took me some trials and errors, but I found that the feature "Glucose" seems to a prominent feature. Here is a couple of SHAP values summary plot.







SHAP is a technique to look at the feature importance of a model, and from my analysis, “Glucose” and its derived features contribute a lot to the model predictions. Therefore, I decided to create this list of features.

```
# Creating interaction features
df2['Glucose_Insulin'] = df2['Glucose'] * df2['Insulin']
df2['Glucose_Age'] = df2['Glucose'] * df2['Age']
df2['SkinThickness_BMI'] = df2['SkinThickness'] * df2['BMI']
df2['Glucose_BMI'] = df2['Glucose'] * df2['BMI']
df2['Glucose_BloodPressure'] = df2['Glucose'] * df2['BloodPressure']
df2['SkinThickness_BMI'] = df2['SkinThickness'] * df2['BMI']

# Creating ratio features
df2['Glucose_BMI_Ratio'] = df2['Glucose'] / df2['BMI']
df2['BMI_squared_if_High_Glucose'] = np.where(df2['Glucose'] > 120, df2['BMI'] ** 2, df2['BMI'])
```

After the created some new features, I oversample the imbalanced label using `RandomOverSampler` (the models perform better so I choose this method).

```
X2 = X.drop(['BloodPressure', 'SkinThickness', 'Insulin'], axis=1)
```

```
# Apply oversampling
ros = RandomOverSampler(sampling_strategy='auto', random_state=42)
X_resampled, y_resampled = ros.fit_resample(X2, y)
X_resampled.shape
```

```
(1000, 12)
```

I also dropped a couple of features, since after some model training, they did not contribute much. I didn't standardize or feature scale the data since we don't have to do that for tree models.

2.3. Model training

Using the similar 70-30 split, I used 3 models Random Forest, Graident Boost and Stacking Classifiers. I have tested on both default hyperparameter and tuning using Random Search, and found that the default hyperparameter does a better job at generalizing on the test set and the hyper tuned models. Hyper-tuned models can often lead to overfitting due the the depth of the trees, at least in my case, so I stick to default hyperparameter of the 3 models used. The evaluation metrics used are accuracy and classification report.

Here is the the output for Random Forest using default hyperparameter.

```
rf = sklearn.ensemble.RandomForestClassifier(random_state=42)
rf.fit(X_train, y_train)

# y pred
y_pred = rf.predict(X_test)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Classification report
report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Test Accuracy: 84.00%
Precision: 84.60%
Recall: 84.00%
F1-Score: 83.92%
```

```
report = classification_report(y_test, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

         0.0       0.89      0.77      0.83       149
         1.0       0.80      0.91      0.85       151

    accuracy                           0.84       300
   macro avg       0.85      0.84      0.84       300
weighted avg       0.85      0.84      0.84       300
```

Here is the output for default hyperparameter for Gradient Boosted Tree (GBT).

```
xgb = XGBClassifier(tree_method='gpu_hist')
xgb.fit(X_train, y_train)

# Predict on test set
y_pred = xgb.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {test_accuracy}")

# Classification report
report = classification_report(y_test, y_pred, output_dict=True)

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
Test Accuracy: 0.8433333039283752
Precision: 85.15%
Recall: 84.33%
F1-Score: 84.23%
```

```
report = classification_report(y_test, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

         0.0       0.90      0.77      0.83       149
         1.0       0.80      0.92      0.86       151

    accuracy                           0.84       300
   macro avg       0.85      0.84      0.84       300
weighted avg       0.85      0.84      0.84       300
```

The GBT does slightly better than RF.

Here is the Stacking Classifier for RF and GBT.

```
# Stacking model
stacking_model = StackingClassifier(
    estimators=[
        ('rf', rf),
        ('xgb', xgb),
    ],
    final_estimator=LogisticRegression(random_state=42),  # /
    cv=5
)
stacking_model.fit(X_train, y_train)

# Y pred
y_pred = stacking_model.predict(X_test)
accuracy = np.mean(y_pred == y_test) * 100
print(f"Test Accuracy: {accuracy:.2f}%")

# Classification report
report = classification_report(y_test, y_pred, output_dict=Tr

precision = report['weighted avg']['precision'] * 100
recall = report['weighted avg']['recall'] * 100
f1_score = report['weighted avg']['f1-score'] * 100

print(f"Precision: {precision:.2f}%")
print(f"Recall: {recall:.2f}%")
print(f"F1-Score: {f1_score:.2f}%")
```

```
[I] [18:09:34.274108] Unused keyword parameter: random_state
Test Accuracy: 85.00%
Precision: 85.44%
Recall: 85.00%
F1-Score: 84.95%
```

```
report = classification_report(y_test, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

         0.0       0.89      0.79      0.84       149
         1.0       0.82      0.91      0.86       151

    accuracy                           0.85       300
   macro avg       0.85      0.85      0.85       300
weighted avg       0.85      0.85      0.85       300
```

The Stacking and GBT have very similar output, the differences are neglectable.

We can see that Stacking and GBT couple with my data handling, does a better job at classifying the diabetic patients than the study.

3. Compare with other notebooks on Kaggle.

I will be comparing my model prediction with golden badge notebooks on [1]. I filtered the search to the most votes, and I will be looking at 3 of them. I will just look at the model predictions and compare to mine.

The first notebook is from **Shruti_Iyyer**. They did not handle the imbalance labels, and their best model only predicts 77%. The 77% might be bias since the label is imbalanced.

The second notebook is from **PIR**. They have quite a sophisticated system to remove outliers, and the final model did 1% better than mine.

The third notebook is from **Vincent Lugat**. They have a very nice visualizations on the how they come up with each feature in feature engineer. The claim they have a 90% accuracy, but they did not split their data set into train and test, this means that their models have a high chance of overfitting.

In conclusion, compare to the study from [2] and the notebook from [1], I think my ML solution does a decent job in classifying the diabetic patients.

4. Video presentation

https://www.youtube.com/watch?v=ns_ZuMEiuKc

References

[1] “Pima Indians Diabetes Database,” *Kaggle*. https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

[2] Md Shamim Reza, R. Amin, R. Yasmin, Woomme Kulsum, and Sabba Ruhi, “Improving diabetes disease patients classification using stacking ensemble method with PIMA and local healthcare data,” *Heliyon (Londen)*, vol. 10, no. 2, pp. e24536–e24536, Jan. 2024, doi: https://doi.org/10.1016/j.heliyon.2024.e24536.