Diabetes Health Indicators Study

Elena Lorite Acosta
University of Alicante

December 9, 2024

Contents

1 Basic

1.1 Review Of The Dataset

The very first thing to do is import the dataset as instructed in the 'Import in python' button at the link of the dataset.

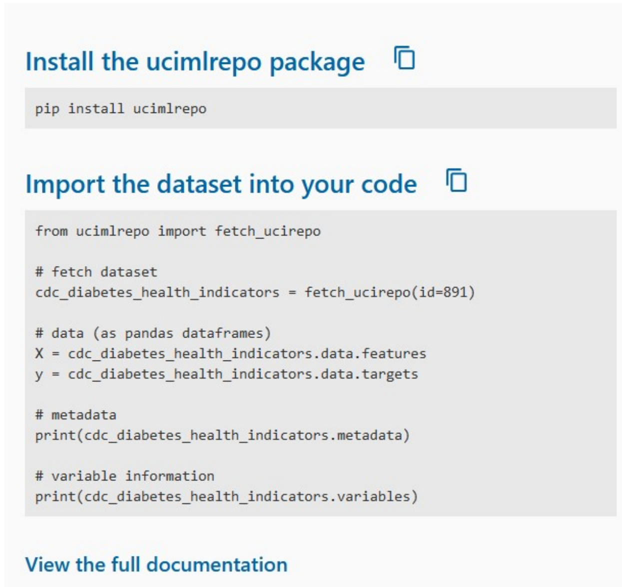

Figure 1: How to import the dataset into the code.

Once the dataset is imported as pandas dataframes (X and y), its contents can be reviewed to check if the dataset is ready for classification. This means it is needed to check if there are variables that aren't numerical, if there are unknown or missing values, or if imbalance exists in the dataset.

To look for categorical features, the instruction `X.info()` is used. This will show all features and their `Dtypes`. Also, the total count of non-null values for each feature is shown. In Figure 2 it is observed that all 21 features are `int64` so all features are numerical. In addition, all features non-null count reaches 253680, which matched the total number of instances the dataset has. To be even more certain, the instruction `X.isnull().sum()` is used to count the total number of null values, and the result is 0 for all categories.

```
Data columns (total 21 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   HighBP                253680 non-null  int64
 1   HighChol              253680 non-null  int64
 2   CholCheck             253680 non-null  int64
 3   BMI                   253680 non-null  int64
 4   Smoker                253680 non-null  int64
 5   Stroke                253680 non-null  int64
 6   HeartDiseaseorAttack  253680 non-null  int64
 7   PhysActivity          253680 non-null  int64
 8   Fruits                253680 non-null  int64
 9   Veggies               253680 non-null  int64
 10  HvyAlcoholConsump     253680 non-null  int64
 11  AnyHealthcare         253680 non-null  int64
 12  NoDocbcCost           253680 non-null  int64
 13  GenHlth               253680 non-null  int64
 14  MentHlth              253680 non-null  int64
 15  PhysHlth              253680 non-null  int64
 16  DiffWalk              253680 non-null  int64
 17  Sex                   253680 non-null  int64
 18  Age                   253680 non-null  int64
 19  Education             253680 non-null  int64
 20  Income                253680 non-null  int64
dtypes: int64(21)
```

Figure 2: Information about each feature in the dataset.

Even though it was already demonstrated that there aren't missing values, there can be values that could not be possible for some features like negative numbers in 'Age'. To check all values are correct, two instructions are used: `X[column].unique()` and `X.describe()`. The first one shows unique values for every feature (Figure 3) and the second one shows a summary of statistics for all features (Figure 4). After checking the outputs, it seems that the dataset is indeed in order and no modifications are needed.

```
Unique values:

HighBP: [1 0]
HighChol: [1 0]
CholCheck: [1 0]
BMI: [40 25 28 27 24 30 34 26 33 21]
Smoker: [1 0]
Stroke: [0 1]
HeartDiseaseorAttack: [0 1]
PhysActivity: [0 1]
Fruits: [0 1]
Veggies: [1 0]
HvyAlcoholConsump: [0 1]
AnyHealthcare: [1 0]
NoDocbcCost: [0 1]
GenHlth: [5 3 2 4 1]
MentHlth: [18  0 30  3  5 15 10  6 20  2]
PhysHlth: [15  0 30  2 14 28  7 20  3 10]
DiffWalk: [1 0]
Sex: [0 1]
Age: [ 9  7 11 10  8 13  4  6  2 12]
Education: [4 6 3 5 2 1]
Income: [3 1 8 6 4 7 2 5]
```

Figure 3: Some unique values for every feature.

	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	HeartDiseaseorAttack	PhysActivity	Fruits	Veggies	...
count	253680.000000	253680.000000	253680.000000	253680.000000	253680.000000	253680.000000	253680.000000	253680.000000	253680.000000	253680.000000	...
mean	0.429001	0.424121	0.962670	28.382364	0.443169	0.040571	0.094186	0.756544	0.634256	0.811420	...
std	0.494934	0.494210	0.189571	6.608694	0.496761	0.197294	0.292087	0.429169	0.481639	0.391175	...
min	0.000000	0.000000	0.000000	12.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...
25%	0.000000	0.000000	1.000000	24.000000	0.000000	0.000000	0.000000	1.000000	0.000000	1.000000	...
50%	0.000000	0.000000	1.000000	27.000000	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000	...
75%	1.000000	1.000000	1.000000	31.000000	1.000000	0.000000	0.000000	1.000000	1.000000	1.000000	...
max	1.000000	1.000000	1.000000	98.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	...

8 rows × 21 columns

Figure 4: Summary of the dataset as in statistics of all features.

Finally, the only thing left to check is if class imbalance exists. Class imbalance means that one target class is much more prevalent than the other. To check this, the trivial system hit percentage is calculated by first calculating the target distribution (percentage) of each class, and then the hit percentage is the maximum distribution (of the most common class). The target distribution is the following:

- 86.0667 % for class 0 (healthy).
- 13.9333 % for class 1 (diabetic or prediabetic).

This basically means that there are way more instances of class 0 than class 1, so there is class imbalance, and the hit percentage is 86.07%. The way this is calculated is shown below:

Listing 1: Computation of target distribution and hit percentage of a trivial system.

```python
# Target distribution
target_distribution = y.value_counts(normalize=True) * 100  # Class percentages
print('Target variable distribution:')
print(target_distribution)

# Trivial system hit percentage (% of most common class)
trivial_hit_rate = max(target_distribution)
most_common_class = y.value_counts().idxmax()  # Class label with the highest count
print(f'\nTrivial system hit percentage: {trivial_hit_rate:.2f}%')
print(f'Corresponding to class {most_common_class[0]}')
```

1.2 Applying Different Classifiers

The next step in this section is to apply some classifiers, apart from the baseline `DummyClassifier()`, to obtain the value of the are under the ROC curve (AUC) and analyze the results. For the basic level only 2 classifiers are needed, in following sections more classifiers (3 more) will be implemented. The classifiers (available in `scikit-learn`) selected are:

- **Logistic Regression** Classifier. Linear classifier, `LogisticRegression()`.
- **Random Forest** Classifier. Tree-based ensembled method, `RandomForestClassifier()`.

As the models are evaluated through the 10-fold cross-validation strategy, it is essential that the 10 partitions used in the process remain identical. To create the partitions, the `StratifiedKFold()` class from `scikit-learn` is used like the professor suggested. Then, for all the classifiers the same partition is passed as an argument to ensure that the models are compared correctly. The models, with the parameters already tuned by hand, are the following:

Listing 2: Definition of different classifiers.

```python
# StratifiedKFold with 10 splits
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# Dummy Classifier (baseline)
dummy_clf = DummyClassifier(strategy="most_frequent", random_state=42)

```

```
# Logistic Regression
logreg_clf = LogisticRegression(solver='lbfgs', max_iter=1000, class_weight='balanced',
    random_state=42)

# Random Forest
rf_clf = RandomForestClassifier(n_estimators=100, max_depth=20, class_weight='balanced',
    random_state=42)
```

Previously it was stated that there is class imbalance, so to take this into account, the parameter `class_weight` was set to 'balance' in the classifiers. This assigns higher weights to the minority class and lower weights to the majority class during training. The following thing to do is to train and evaluate all three models to get the AUC values. There is a function that does this, `cross_val_score()`, which evaluate a score by cross-validation. To ensure that the score is the area under the ROC curve, the parameter `scoring` is set to 'roc_auc'.

Listing 3: Evaluation of a score by cross-validation.

```
dummy_auc = cross_val_score(dummy_clf, X, y, cv=skf, scoring='roc_auc')

logreg_auc = cross_val_score(logreg_clf, X, y.values.ravel(), cv=skf, scoring='roc_auc')

rf_auc = cross_val_score(rf_clf, X, y.values.ravel(), cv=skf, scoring='roc_auc')
```

The value for AUC ranges from 0 to 1. A model that has an AUC of 1 is able to perfectly classify observations into classes while a model that has an AUC of 0.5 does no better than a model that performs random guessing. The higher the AUC value the better while an AUC score of 0.5 is no better than a model that performs random guessing.

Table 1: Comparison of basic classifiers.

	Dummy Classifier	**Logistic Regression**	**Random Forest**
AUC (mean)	0.5	0.8225	0.8042
Processing Time (s)	0.4234	142.57	268.83

After running all classifiers, the results can be seen in Table 1. The AUC values obtained with the two classifiers are way better than the one obtained with the Dummy classifier, as expected. The time spent running each instruction in Listing 3 is also shown. While the Dummy classifier is instant, both the Logisitic Regression and the Random Forest classifiers take some time due to the large size of the dataset. Also, the parameters were tuned by hand, so probably searching for optimal parameters automatically could improve de AUC score. This will be done in the Advanced Level.

2 Intermediate

2.1 Applying More Classifiers

In this section three more classifiers will be applied to compare the results to the baseline and to try and select the best ones. These new classifiers are:

- **Bagging Classifier**. Creates multiple subsets of the dataset with replacement (bootstrapping) and training a base estimator (Decision Tree) on each subset. The results are aggregated to make the final prediction. `BaggingClassifier()`.
- **Gradient Boosting Classifier**. Algorithm that combines several weak learners into strong learners, in which each new model is trained to minimize the loss function. `GradientBoostingClassifier()`.
- **K-Nearest Neighbors**. Implements the k-nearest neighbors strategy, `KNeighborsClassifier()`.

These classifiers are also evaluated using the same 10 partitions as before, and some of their parameters were tuned by hand:

Listing 4: Definition of more classifiers.

```
# Bagging Classifier
bag_clf = BaggingClassifier(n_estimators=100, max_samples=0.8, max_features=1.0 random_state=42)

# Gradient Boosting Classifier
gb_clf = GradientBoostingClassifier(n_estimators=150, learning_rate=0.1, max_depth=3, random_state=42)

# kNN
knn_clf = KNeighborsClassifier(n_neighbors=10, weights='distance')
```

Note that kNN doesn't handle class imbalance inherently, but weighting the neighbors by setting the parameter `weights` to 'distance' can help mitigate imbalance effects. Even so, it is expected to perform worse than the others. The evaluation method is the same as in Listing 3, but using as estimator the new classifiers. The results of these classifiers, along with the previous ones, are shown in Table 2.

Table 2: Comparison of AUC values of 5 different classifiers.

	Logistic Regression	**Random Forest**	**Bagging Classifier**	**Gradient Boosting**	**kNN Classifier**
AUC (mean)	0.8225	0.8042	0.7894	0.8304	0.7504
Processing Time (s)	142.57	268.83	818.07	444.95	1325.86

This values are, again, better than the one obtained by the baseline classifier (0.5). After applying the 5 classifiers, seems like **Logistic Regression**, **Random Forest** and **Gradient Boosting** are the best ones. As testing all the classifiers take too much time (some take over a quarter of an hour), only the best ones will be tested in the future. Automatic parameter tuning will be performed in the Advanced Level to ensure that the classifiers get the maximum AUC value possible.

2.2 Automatic Feature Selection

Automatic feature selection helps identify the most important features in the dataset, which can improve model performance, reduce overfitting, and speed up training. Two different algorithms for automatic feature selection will be tested, and then classification will be performed after that to see if results improve.

- **Tree-Based Model** (Random Forest). This method naturally rank features by importance based on their contribution to splits and decisions.
- **Recursive Feature Elimination** (RFE). RFE works by recursively removing the least important features based on the weights or coefficients of a trained model.

To get the top features with the Tree-Based Model, it is necessary to first train the model , in this case a Random Forest, with all the features to determine its importance. Then the importance scores will be used to sort the dataset in descending order, to finally select the top 12 features and reduce the dataset.

Listing 5: Feature selection using Random Forest.

```
# Train Random Forest
rf_selector = RandomForestClassifier(class_weight='balanced', random_state=42)
rf_selector.fit(X, y.values.ravel())

# Get feature importance scores
rf_feature_importances = rf_selector.feature_importances_
rf_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance': rf_feature_importances})
rf_importance_df = rf_importance_df.sort_values(by='Importance', ascending=False)

# Select top 12 features
rf_top_features = rf_importance_df.head(12)['Feature'].values
X_rf_reduced = X[rf_top_features]   # Reduced dataset
print("Top Features Selected:", rf_top_features)
```

Getting the top features with RFE is easier, as `scikit-learn` already has a class that implements RFE. First it is necessary to initialize the model, Logistic Regression in this case, and then apply the `RFE` class to get the top features. Finally the result can be transformed to a reduced dataset.

Listing 6: Feature selection using RFE.

```
# Initialize the model
logreg_selector = LogisticRegression(max_iter=1000, class_weight='balanced', random_state
    =42)

# Apply RFE - Select top 12 features
rfe = RFE(estimator=logreg_selector, n_features_to_select=12)
X_rfe = pd.DataFrame(rfe.fit_transform(X, y.values.ravel()))  # Reduced dataset

# Check selected features
selected_features = X.columns[rfe.support_]
print("Selected Features:")
print(selected_features.tolist())
```

Table 3: Feature selection comparison.

	Features Selected
Tree-Based Model	BMI, Age, GenHlth, Income, HighBP, HighChol, Fruits, Smoker, DiffWalk, PhysHlth, Education, MentHlth
RFE	BMI, Age, GenHlth, Income, HighBP, HighChol, Fruits, CholCheck, Stroke, HeartDiseaseorAttack, HvyAlcoholConsump, Sex

2.3 Evaluation Of Classifiers After Feature Selection

The features selected by both methods can be seen in Table 3. This features will be the ones used to evaluate again the top classifiers (LR, RF and Boosting Gradient) with `cross_val_score()`, but using the reduced datasets instead of the original one. The results for both methods and for the three classifiers are compared in Table 4.

Table 4: Comparison of model performance after applying feature selection algorithms.

	Logistic Regression	**Random Forest**	**Gradient Boosting**
AUC (Tree-Based Model)	0.8174	0.7902	0.8247
AUC (RFE)	0.8221	0.7803	0.8295

Table 5: Processing time for each model after applying feature selection algorithms.

	Logistic Regression	**Random Forest**	**Gradient Boosting**
Time (s) (RF)	49.50	247.99	355.54
Time (s) (RFE)	41.31	200.32	372.27

Even though the AUC scores didn't improve, the processing time was significantly reduced (Table 5). Some classifier took approximately half the time to evaluate, like Logistic Regression. This is because the features were reduced by half.

2.4 Wilcoxon Signed-Rank Test

To determine if results are better or equivalent from one another, the Wilcoxon Test will be used. The library `SciPy` provides a function that calculate the Wilcoxon signed-rank test, `wilcoxon`. First, AUC scores from the same classifier but different feature selection methods will be used to perform the test. This means that three tests will take place, one for each classifier.

The p-value returned by the Wilcoxon test indicates whether the distributions are different or not, as it examines two tails. If the p-value is less than the significance level (0.05) then the null hypothesus is rejected and determine that the feature selection method significantly impacts the classifier’s performance. If the p-value is greater than or equal to the significance level, there is no significant difference. Using the ’greater’ parameter, it is possible to determine if one method is better than the other one (the comparison is if ’x’ is greater than ’y’).

In the tests performed, first it was only tested if the feature selection method has an impact on the AUC scores, this is call a **two-sided test**, and is done by using the default value for the parameter `alternative`.

Listing 7: Wilcoxon test to determine if the feature selection method impacts the results.

```python
# Perform test for logreg
logreg_wilcox_W, logreg_p_value =  wilcoxon(rf_logreg_auc, rfe_logreg_auc, correction=True)
print('Results of the Wilcoxon test for logreg')
print(f'Wilcox W: {logreg_wilcox_W}, p-value: {logreg_p_value:.4f}')

# Perform test for rf
rf_wilcox_W, rf_p_value =  wilcoxon(rf_rf_auc, rfe_rf_auc, correction=True)
print('Results of the Wilcoxon test for rf')
print(f'Wilcox W: {rf_wilcox_W}, p-value: {rf_p_value:.4f}')

# Perform test for gb
gb_wilcox_W, gb_p_value =  wilcoxon(rf_gb_auc, rfe_gb_auc, correction=True)
print('Results of the Wilcoxon test for gb')
print(f'Wilcox W: {gb_wilcox_W}, p-value: {gb_p_value:.4f}')
```

Table 6: Results of the two-sided Wilcoxon test.

	Logistic Regression	**Random Forest**	**Gradient Boosting**
p-value	0.002	0.002	0.002

The p-value after the two-sided test indicates that indeed the method for feature selection has an impact in the results. To find out which one is better, simply looking at the AUC (mean) values in Table 4 can be determined. But if we want to be really sure, the Wilcoxon test can be performed as a **one-sided test** by setting the parameter `alternative` to ’greater’.

Table 7: Results of the one-sided Wilcoxon test.

	Logistic Regression	**Random Forest**	**Gradient Boosting**
p-value	1.0	0.001	1.0

In Table 6 the p-value of the two-sided Wilcoxon test indicated that the method for feature selection affected the AUC values obtained. In Table 7 the p-value of the one-sided Wilcoxon test indicates that the tree-based method is only better in the case of the Random Forest classifier, while the RFE is better for the other two classifiers. Again, looking at the mean of the AUC values this corroborates what the Wilcoxon test stated. As **RFE** seems to get better results, this is the method that will be used in future tests.

3 Advanced

3.1 Optimizing Parameters Automatically

In previous sections, the classifiers where tuned by hand. This takes too much time and effort, and the parameters are probably not even the most optimal. To address this problem, there are algorithms that automatically search for optimal parameters. This is known as **automated hyper-parameter tuning**, choosing a set of optimal hyperparameters for a learning algorithm automatically. There are several algorithms and methods that do that, but there are two that are very commonly used when working with `scikit-learn`.

- **Grid Search**. Exhaustively examines all combinations of hyperparameters.
- **Random Search**. Samples various hyperparameters from a distribution.

In this project only the Random Search algorithm, `RandomizedSearchCV`, will be used, as is way less computational expensive than the Grid Search. This method basically follows four basic steps:

1. Define the hyperparameters to search over (Python dictionary).
2. Set a lower and upper bound on the values (if it's a continuous variable) for each hyperparameter.
3. A random search randomly samples from these distributions a total of N times, training a model on each set of hyperparameters
4. The hyperparameters associated with the highest accuracy model are selected.

The k-fold-cross-validation procedure can be used when optimizing the hyperparameters and when comparing and selecting a model. An optimistically biased assessment of the model's performance is likely to result from using the same cross-validation process and dataset for both model selection and tuning. To overcome this bias, a **nested cross-validation** procedure can be done. Nested cross-validation is a good approach for hyperparameter tuning because it helps prevent overfitting during the model selection process and provides an unbiased estimate of the model's performance on unseen data. To do this, the Random Search must perform **inner** cross-validation and to evaluate the model's performance it is required **outer** cross-validation. A function that does all the procedure has been created so that it can be called when trying different classifiers. This function is `hyperparameter_tuning()` and has been created following the tutorials in [1] and [2]. This approach basically sets an outer cross-validation of 10 folds to match the original setup, and an inner cross-validation of 3 folds to perform parameter tuning. The inner folds are typically smaller for efficiency.

Listing 8: Function to apply the nested cross-validation and Random Search procedures to any classifier.

```
def hyperparameter_tuning(model, param_dist, X, y, model_name):
  # Inner CV for hyperparameter tuning
  cv_inner = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
  random_search = RandomizedSearchCV(estimator=model, param_distributions=param_dist,
    scoring='roc_auc', n_iter=20, cv=cv_inner, random_state=42, verbose=1)
  random_search.fit(X, y.values.ravel())

  # Outer CV for model evaluation
  outer_cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
  scores = cross_val_score(random_search,X,y.values.ravel(),cv=outer_cv,scoring='roc_auc')

  # Report results
  print(f"\nBest params for {model_name}: {random_search.best_params_}")
  print(f"Nested CV {model_name} AUC (mean): {scores.mean():.4f}, Std: {scores.std():.4f}")
```

Listing 9: Random Search method to optimize parameters for the Random Forest classifier.

```
# Define the hyperparameter distribution
dist_rf = {
    'n_estimators': randint(50,200),      # Number of trees
    'max_depth': [None, 10, 20, 30],      # Maximum depth of trees
    'min_samples_split': randint(2, 20),  # Range for splits
    'min_samples_leaf': randint(1, 15),   # Range for leaf samples
    'max_features': ['auto', 'sqrt', 'log2']
}

rf = RandomForestClassifier(random_state=42, class_weight='balanced')
hyperparameter_tuning(rf, dist_rf, X_rfe, y, 'Random Forest')
```

To show how a classifier can be passed to the function in Listing 8, the code for the Random Forest classifier is shown (Listing 9). For the rest of the classifiers is mostly the same, changing the hyper-parameter distributions for the Random Search. Note that the Random Search process is performed on the reduced dataset obtained after applying the RFE method, this is to accelerate the process.

Table 8: Best parameters obtained for each classifier after Random Search.

	Logistic Regression	Random Forest	Gradient Boosting
Best Parameters	C = 4.4593, penalty = 'l2', solver = 'lbfgs'	max_depth = 10 max_features = 'log2' min_samples_leaf = 5 min_samples_split = 10 n_estimators = 167	learning_rate = 0.0928 max_depth = 4 min_samples_leaf = 5 min_samples_split = 18 n_estimators = 78
Processing Time (min)	75	202	170

Table 8 shows the best parameters found by the Random Search process for each classifier. Observing the processing time, it is clear that these methods are computationally and time consuming. If the grid search method had been used, the results and parameters obtained would be betterand more optimized, in exchange for using more computational resources and time. After optimzing the parameters the AUC scores obtained for each classifiers are shown in Table 9.

Table 9: AUC values after automatic parameter optimization.

	Logistic Regression	Random Forest	Gradient Boosting
AUC (mean)	0.8221	0.8272	0.8294

4 Final Prediction System

The last step of the project is to choose the classifier that will be part of the final predictive system. To select it, the results obtained throughout the project have been summarized in Table 10. After analyzing them, the chosen classifier was **Random Forest** (RFE-Reduced Dataset and parameters automatically optimized). This is the chosen one because the AUC scores obtained by Random Forest after tuning correctly the parameters improved a lot in comparison to the classifier when using hand tuned parameters. It has been chosen over Gradient Boosting because, although the latter obtains slightly better AUC scores, Random Forest directly addresses the class imbalance, whereas Gradient Boosting does not.

Table 10: Comparison of the results (AUC values) obtained during the project.

Classifier	Dataset Type	Parameter Tuning	Best AUC
Logistic Regression	Full Dataset	Hand Tuned	0.8225
	RF-Reduced Dataset	Hand Tuned	0.8042
	RFE-Reduced Dataset	Hand Tuned	0.8221
		Auto Tuned	0.8221
Random Forest	Full Dataset	Hand Tuned	0.8042
	RF-Reduced Dataset	Hand Tuned	0.7902
	RFE-Reduced Dataset	Hand Tuned	0.7803
		Auto Tuned	0.8272
Gradient Boosting	Full Dataset	Hand Tuned	0.8304
	RF-Reduced Dataset	Hand Tuned	0.8247
	RFE-Reduced Dataset	Hand Tuned	0.8295
		Auto Tuned	0.8294

A final evaluation of the model selected is performed to conclude the project. First the best hyperparameters (from Random Search) are used initialize the model and then 10-fold cross-validation is performed on the Random Forest Classifier. Then the most influential features are highlighted to know the most relevant indicators that affect health. Finally, after running the process mentioned, the **AUC score** mean is **0.8272** and the feature importance can be observed in Figure 5.

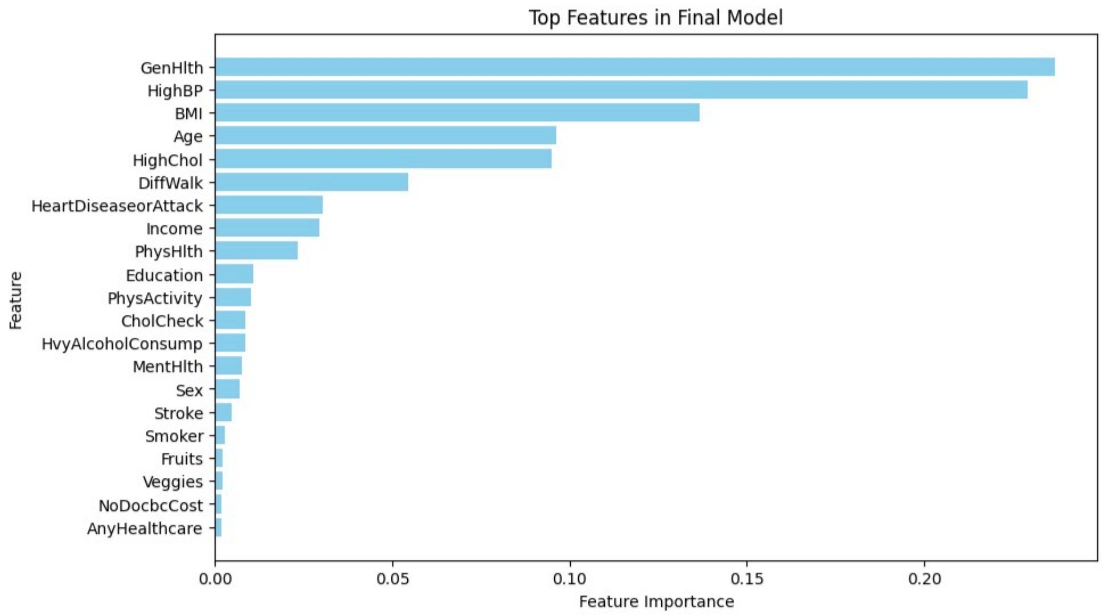

Figure 5: Most relevant indicators that affect health.

5 Conclusions

In this project a predictive system for diabetes classification using health indicators was developed, employing diverse machine learning algorithms and rigorous evaluation techniques. Starting with data preprocessing and handling class imbalance, feature selection methods were explored, such as Random Forest-based selection and Recursive Feature Elimination, to enhance interpretability and model performance. Various classifiers, including Logistic Regression, Random Forest, Gradient Boosting, and others, were implemented, with hyperparameters tuned using Randomized Search and nested cross-validation to ensure robust evaluation.

The project highlights the importance of feature selection, hyperparameter optimization, and addressing class imbalance to build reliable predictive models. This system tries to guide healthcare interventions by identifying key health indicators influencing diabetes, providing a foundation to design intervention plans or data-driven medical recommendations.

References

[1] Jason Brownlee. Nested Cross-Validation for Machine Learning with Python - MachineLearningMastery.com — machinelearningmastery.com. https://machinelearningmastery.com/nested-cross-validation-for-machine-learning-with-python/. [Accessed 04-12-2024].

[2] DamenC. A Guide to Nested Cross-Validation with Code Step by Step — cd_24. https://medium.com/@cd_24/a-guide-to-nested-cross-validation-with-code-step-by-step-6a8ad06d5af2. [Accessed 04-12-2024].

[3] GeeksforGeeks. Feature selection in python with scikit-learn. https://www.geeksforgeeks.org/feature-selection-in-python-with-scikit-learn, Jun 2024.

[4] Scikit-Learn. sklearn.ensemble — scikit-learn.org. https://scikit-learn.org/stable/api/sklearn.ensemble.html. [Accessed 03-12-2024].

[5] Adrian Rosebrock. Introduction to hyperparameter tuning with scikit-learn and Python - PyImageSearch — pyimagesearch.com. https://pyimagesearch.com/2021/05/17/introduction-to-hyperparameter-tuning-with-scikit-learn-and-python/. [Accessed 04-12-2024].