1. **Problem description:**

Create a classifier using a dataset of 47180 patients to train the model.

Each patient will have 27 attributes:

* Gender: Gender of the patient (e.g., Male or Female).
* Age: Age of the patient.
* LOSdays: Length of Stay in days.
* admit\_type: Type of admission (e.g., EMERGENCY, ELECTIVE).
* admit\_location: Location from which the patient was admitted.
* AdmitDiagnosis: Diagnosis upon admission.
* Insurance: Type of insurance the patient has.
* Religion: Patient's religion.
* marital\_status: Marital status of the patient.
* ethnicity Ethnic background of the patient.
* NumCallouts: Number of callouts.
* NumDiagnosis: Number of diagnoses.
* NumProcs: Number of procedures.
* AdmitProcedure: Procedure upon admission.
* NumCPTevents: Number of CPT events.
* NumInput: Number of inputs.In a hospital or medical context, "input" typically refers to the amount of fluids, medications, or nutrients given to a patient.
* NumLabs: Number of lab tests.
* NumMicroLabs: Number of microbiology labs.
* NumNotes: Number of notes.
* NumOutput: Number of outputs.
* NumRx: Number of prescriptions.
* NumProcEvents: Number of procedure events.
* NumTransfers: Number of transfers.
* NumChartEvents: Number of chart events.
* ExpiredHospital: Indicates if the patient expired in the hospital (0 for No, 1 for Yes).
* TotalNumInteract: Total number of interactions.
* LOSgroupNum: Group number based on the length of stay.

The classifiers will then be given another dataset of patients with the “ExpiredHospital” column being unknown. Using the other 26 attributes of the patient, the classifiers will have to predict whether this patient has expired in hospital or not.

1. **Data preprocessing and transformations:**

Here are all of the data preprocessing I did for the classifiers:

* Dropping the *‘LOSgroupNum’* column:

Because 42447 out of 47180 values in this column are missing, the importance of this feature in the training process is minimal. Furthermore, because of how little information we have on this feature, using it during the training process can make the classifiers overfit. Therefore, I thought it would be better to drop it altogether.

This line of code below is used to remove the feature from the dataset.



* Filling missing values in the dataset:

Because the dataset has two types of data: numerical and categorical, and both types have missing values. Therefore, I had to employ two approaches:

* + For numerical features: The missing numbers in a feature are filled with the average (the mean) value of that feature.

The code:



Where x1 are the numerical features.

* + For categorical features: The missing data in a feature are filled with the most frequent class (the mode) of that feature.

The code:



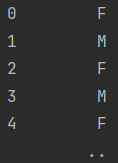
Where x2 is are the categorical features.

* Encoding categorical features:

Because classifiers can’t interpret human language, classes that are written in words/letters will have to be encoded into numbers. Where each number will represent a class of that feature. This can be done using the .factorize() function from the *pandas* package.



A black background with white numbers

Description automatically generatedBefore and after encoding:

* Normalization for numerical features:

Some models like Support Vector Classifier (SVC) or Neural Network (NN) would perform better when the data in the numerical features are normalized. Here are some reasons why:

* + Because the scale of the numbers varies a lot depending on the feature. Normalization will scale all of the numbers to the same range, so the model can’t be biased towards features with larger numbers.
  + Because this dataset has a lot of outliers, normalization reduces the emphasis placed on those outliers and prevents overfitting.
  + I used the *StandardScaler()* from the *sklearn.preprocessing* package for the normalization.

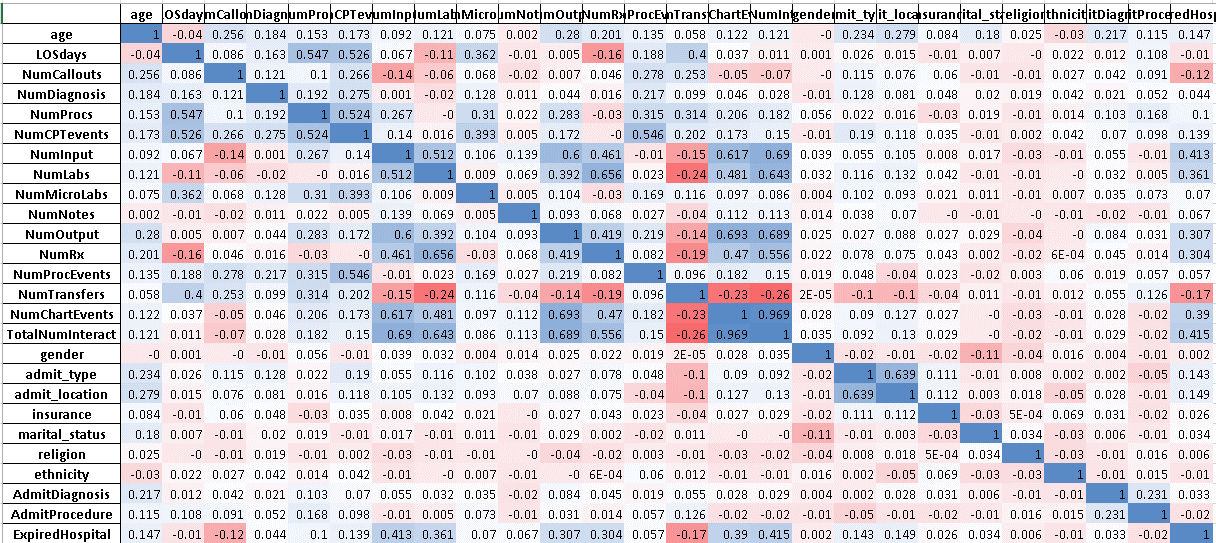


A black background with white text

Description automatically generated

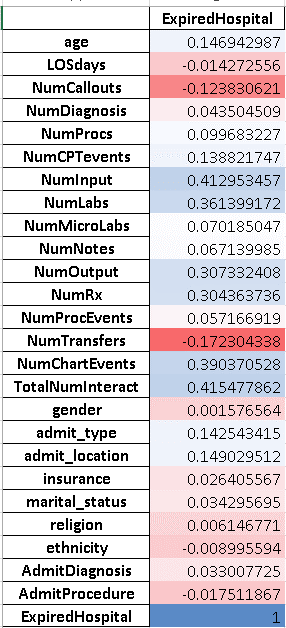
* Feature importance:

The correlation table of the features in the data.



Some interesting points:

* + Most of the features are not highly correlated, with the exception of “TotalNumberInteract” – “NumChartEvents”. That also means most of the features are not redundant, so we don’t have to remove any features due to similarity.
  + We can see the correlations between the target feature (“ExpiredHospital”) with the other feature:



As illustrated, some features have a much stronger correlation to “ExpiredHospital” than others. Therefore, feature importance will vary a lot depending on the feature.

Some features with very low correlation like “gender” can be removed entirely if the classifier can’t handle low-importance features well.

* Lastly, I used an 80-20 split to generate the train and test set for the classifiers.



1. **Solving the problem:**

I experimented with a lot of classifiers, all of the ones that were taught in this course and some more outside of this course. Then I selected the best performing classifier and start training many models of that classifier, the model with the highest f1\_macro score in a training session will be submitted to Kaggle. To get the best performing model when training a classifier, I used some techniques:

* Changing the weight of the labels: As the dataset is unbalanced (the number of patients that expired is 8.9 times those who didn’t). Therefore, the number of negative labels will be much larger than the number of positive ones. So I had to use this line code to calculate the scale.



And then increase the weight of the positive values by using the ‘scale\_pos\_weight’ parameter.



This worked for all of the classifiers except for NN as NN didn’t have this parameter. Instead, I had to pass the scaling into the loss function of the NN.



* Hyperparameters tuning:

For the classifiers that performed well by default, I started tuning the hyperparameters of those classifiers. These classifiers include: Random Forest, XGBoost and LightGBM. And all 3 of those were tuned using k-folds cross-validation, more specifically:

* + Random Forest and XGBoost were tuned using **RandomizedSearchCV** from the *sklearn.model\_selection* package.



* + LightGBM were tuned using **BayesSearchCV** from the *skopt* package.



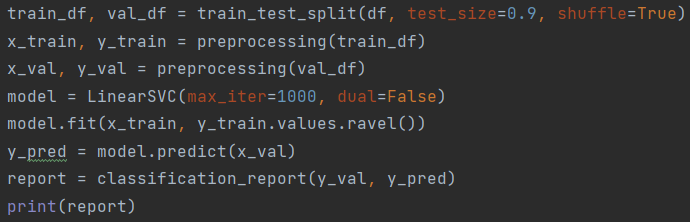
* + In k-fold cross-validation, the dataset is divided into k equal-sized folds. The model is trained on k-1 folds and evaluated on the remaining fold. This process is repeated k times, with each fold being used as the validation set once. The average performance across all k folds is then calculated to give an estimate of how well the model will generalize to new data.
    - For **RandomSearchCV**, we can specify a range of values that each hyperparameter can take. Then the algorithm will randomly choose a set of hyperparameters in that range and starts training and evaluating on the folds. The number of times this process happens is specified by the user. After the search is finished, the model with the best-performing hyperparameters will be returned.
    - For **BayesSearchCV**, the process is almost the same as **RandomSearchCV**. With the only difference is that parameters are not randomly chosen, but rather picked using **Gaussian Process**. This helps to find the optimal set of hyperparameters faster.
    - Details on what parameters were tuned in each classifier will be given in part 4.

1. **Classification techniques used and results:**

Here’s a list of all of the classifiers that I used: Support Vector Classifier (**LinearSVC** from *sklearn.svm*), Random Forest (**RandomForestClassifier** from *sklearn.ensemble*), K-nearest-neighbors (**KNeighborsClassifier** from *sklearn.neighbors*), Neural Network (*pytorch*) and Gradient Boosting Tree (***XGBoost***and ***LightGBM***).

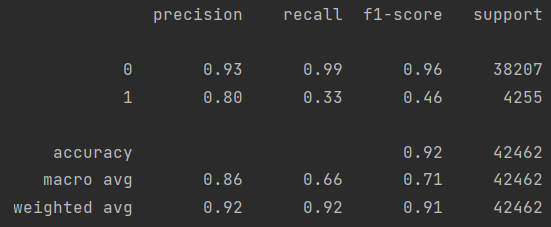
Now I’ll go through the code and results of each classifier in the order of how well they performed (worst to best):

* **LinearSVC:**
  + The code:



I experimented with different train-test sizes, and using only 10% of the data to train seems the give me the best results on this classifier.

* + The results:

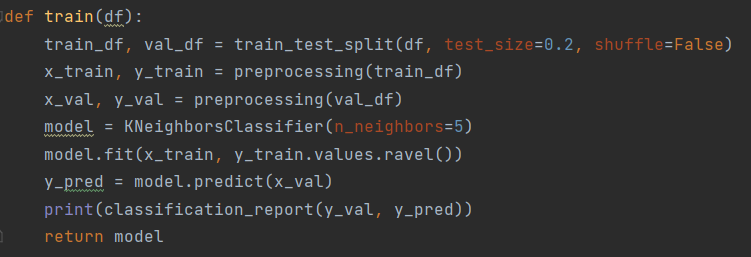


As you can see from the results, this model only scored 0.71 on the f1 macro score, this could be because I didn’t bother optimizing the classifier, mainly because the default model didn’t do as well as the ensemble classifiers. So I thought my time would be better spent on experimenting with those.

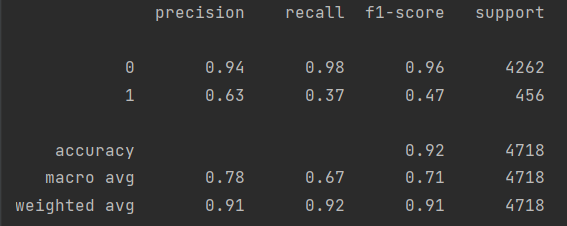
* **K-nearest-neighbors:**

KNN performed about the same as LinearSVC, which is not enough to be considered good. There didn’t seem to be any parameters that I could tune aside from choosing the number of neighbors. But after some trial and error, the optimal number of neighbors still performed quite poorly.

* + The code:



* + The results:

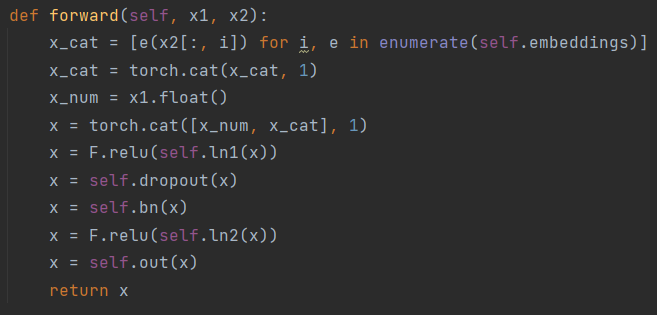


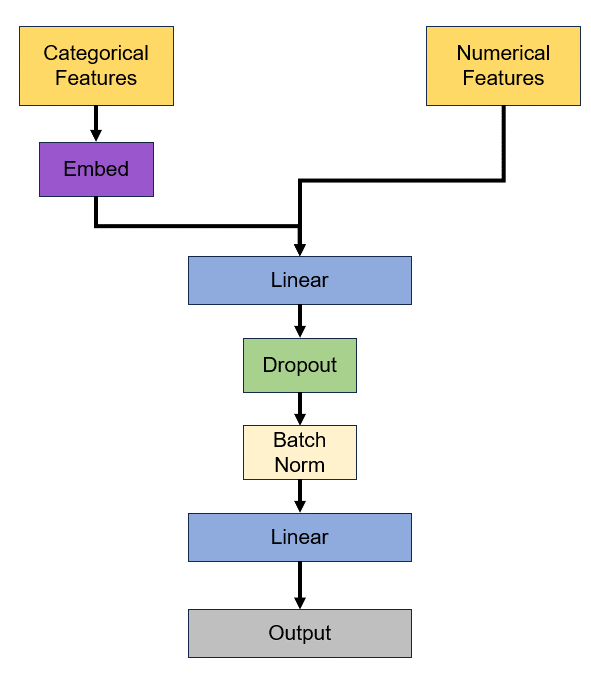
The best model scored only 0.71 on f1 macro, and most models get scored 0.7, which is about as good as LinearSVC.

* **Neural Network:**

This classifier performance was underwhelming as well, mainly because I’m quite inexperienced with neural networks, and I don’t think neural networks can perform too well on this dataset. Though, it did yield some results.

* + First, I want to talk about the structure of the NN. Here’s the code and a graph detailing the structure of my NN:





To elaborate on some part of the structure:

* + - The activation function for the linear layers is the **Rectified Linear Unit (ReLU)** function.
    - The input is split into categorical and numerical.
    - Because the NN can’t interpret categorical data even when encoded, I had to run the categorical features through a process called “embedding”. Embedding is essentially turning different classes in a feature into vectors of numbers, making it possible for the NN to interpret.
    - After the NN has finished embedding the categorical features, both types of features will be combined into a single input.
    - The concatenated input will go through a linear layer and a dropout layer. The dropout layer randomly drops a certain percentage of the input (in this case, 20% of the input). This is to prevent the NN from overfitting.
    - After dropping out, the data will undergo a process called “batch normalization”. Batch normalization has several benefits, including faster convergence during training, improved generalization performance, and reduced sensitivity to hyperparameters.
    - Lastly, the input will go through another linear layer, and then finally output.
  + Secondly, I want to go through the training process of the NN:

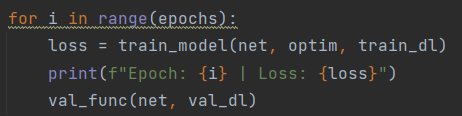
I used the **Adam** optimizer in the *torch* package for my NN.



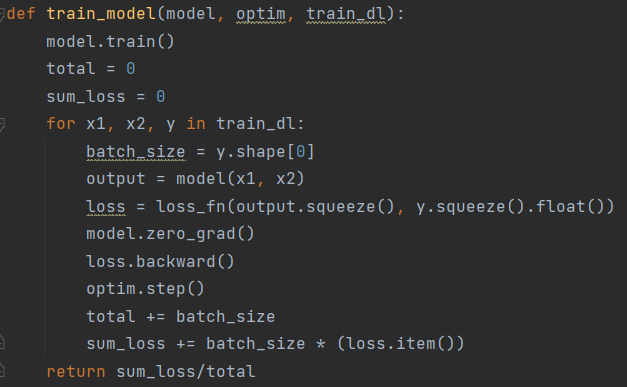
The learning rate starts at 0.015 but will decay as the training goes on.

The NN was trained through 200 epochs.





The train function:



Simply put, this function performs the forward and backpropagation phase of the NN and returns the loss.

The loss function of this NN is **BCEWithLogitsLoss** from the *pytorch* library.



This function combines a Sigmoid layer and the Binary Cross Entropy Loss function.

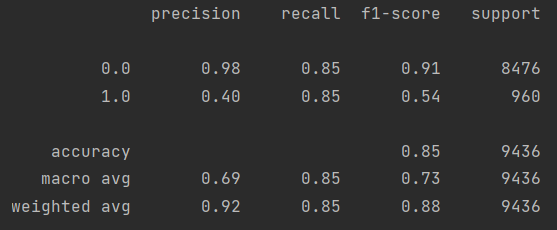
The validation function:

A screen shot of a computer program

Description automatically generated

This function reports the results of the model after each epoch.

* + Results:



We can see that the NN didn’t do too well on the F1 macro score, but it at least performed better than the LinearSVC and KNN. This could be because I didn’t spend too much time on the NN as I didn’t think it could perform as well as other classifiers.

* **Random Forest:**

Process of making the Random Forest Classifier:

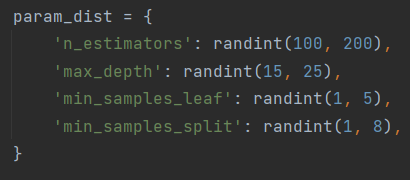
1. Initialization:

After preprocessing the data, I initialized the **RandomForestClassifier** with a class\_weight parameter to deal with the unbalanced dataset.

A black background with white text

Description automatically generated

1. Hyperparameters tuning:



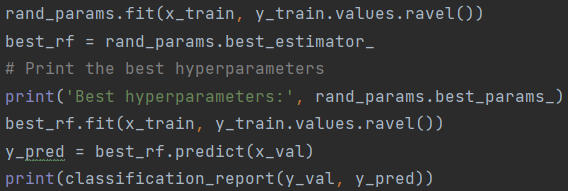
A computer screen shot of a code

Description automatically generated

I used **RandomizedSearchCV** to find the most optimal parameters for the Random Forest.

* + For this classifier, I only tuned two parameters:
    - *n\_estimators*: The number of trees to be used in the model.
    - *max\_depth*:The maximum depth a tree can reach.
    - *min\_samples\_leaf*: The minimum number of samples required at each leaf node.
    - *min\_samples\_split*: The minimum number of samples required to split a node.
  + The parameter distribution specifies the range of the parameter that the randomized search can choose. E.g:

*‘n\_estimators’: randint(50, 200)* means the search can choose a random value between 50 and 200 to be the number of trees used.



* + After training and evaluating 50 randomly chosen sets of parameters on 5 folds. The **RandomizedSearchCV** will return the model with the best-performing hyperparameters.

A screen shot of a computer code

Description automatically generated

* + We can then train the best model with our data and evaluate based on the prediction.

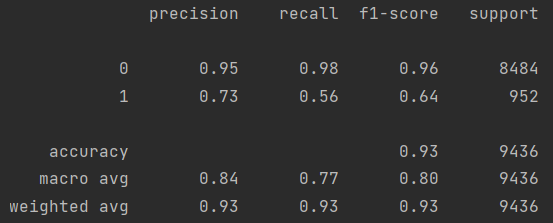
1. Results evaluation:

I will run step 1 and 2 multiple times, and see what the optimal parameters would be, then I would tweak the range of the parameter distributionto be closer to the optimal parameters. This is to make the new model performs closer to the best model in the last iteration.

After doing this a few times, this is the set of hyperparameters that I got:



And the results of this model:



As can be seen from the image, this classifier performed much better than the 3 classifiers above, scoring 0.8 on f1 macro.

This was also the first model that got me a decent score on Kaggle as well.



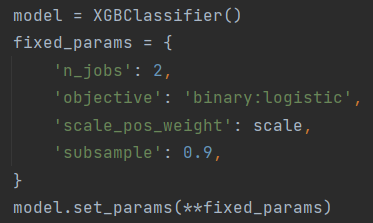
* **XGBoost:**

The steps in making an XGBoost classifier is similar to making the Random Forest classifier. But I spent more time on this classifier than the Random Forest because the default model of this one did better than that of the Random Forest.

1. Initialization:



Importing the classifier.

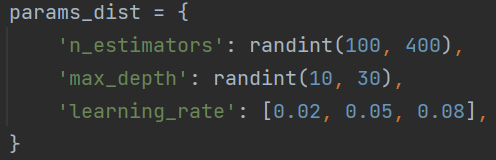


Initializing the **XGBClassifier** and set up some parameters:

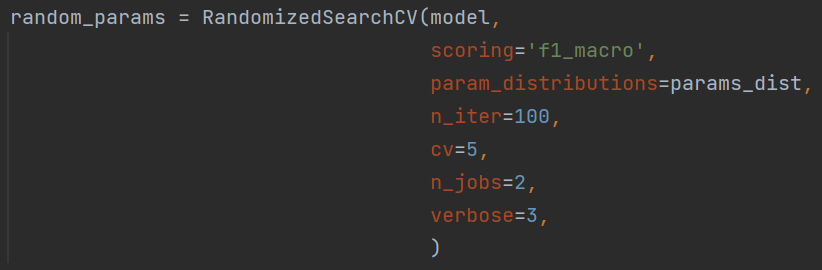
* + *‘n\_jobs’: 2* – gives the classifier more CPU power to make the process faster.
  + *‘objective': 'binary:logistic’* – because we’re doing binary classification.
  + *'scale\_pos\_weight': scale* – to address the unbalanced dataset.
  + *‘subsample’: 0.9* – to prevent the model from overfitting.

1. Hyperparameters tuning:

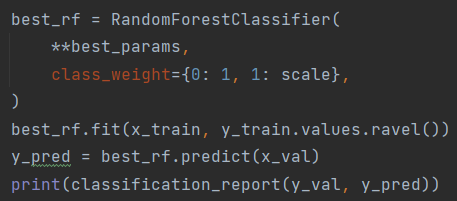
The parameter distribution:



Because I knew I was going to spend more time on this classifier, I also tuned the learning rate of the model.



Initializing the **RandomizedSearchCV**. The scoring metric will be “f1\_macro” and the search will run for 100 iterations. After finishing the search, the best parameters will be returned.

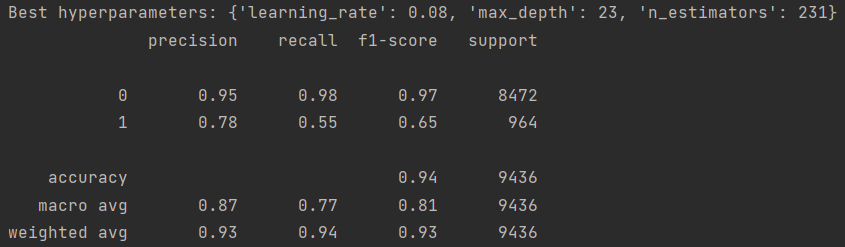


A new Random Forest model will be fit with the best parameters and be evaluated.

1. Results evaluation:

Similar process to the Random Forest, where the parameter distribution will be tweak to be closer to the parameters that performed well.

After running multiple iterations of the search, we can get a model with good results like:



The f1 macro score on this classifier in 0.81, this might not seem like a big performance improvement over Random Forest, but we have gotten to the point where a 0.1 difference is very big.

This was also the classifier I used to get a good score on Kaggle for a long time, so I submitted dozens of models of XGBoost. Some notable submissions:



* **LightGBM:**

Lastly, this is the classifier that performed the best for my case, and it was also the one that I spent the most time on.

The steps to make a a LightGBM classifier is similar to XGBoost and Random Forest, only this time I experimented with 2 different parameters optimizing technique: Optuna and BayesSearchCV.

Since the Optuna LightGBM did worse, I’m going to go through that one first.

**LightGBM with Optuna:**

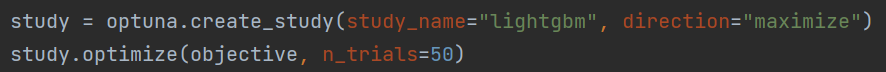
1. Initialization:

Import *optuna* and *lightgbm:*





Creating the optuna study:

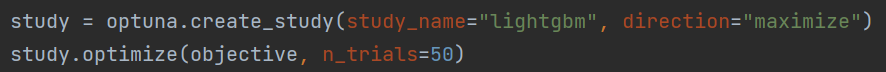


1. Hyperparameters tuning:

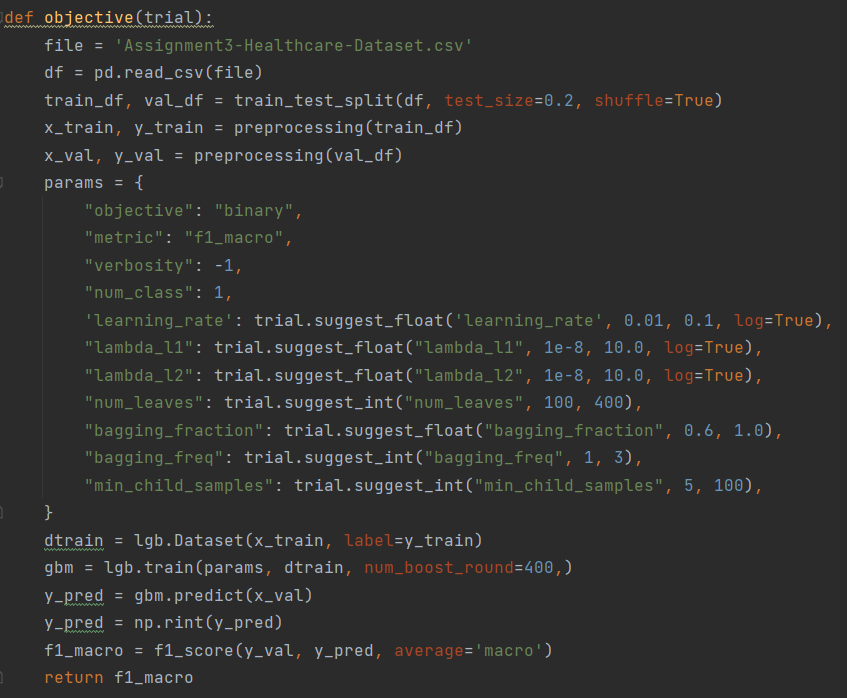
*Optuna* is a Python library that automates the hyperparameter optimization process of machine learning models.

The basic idea behind *Optuna* is to provide a space of hyperparameters to test in order to determine the combination of hyperparameters that optimize your model.

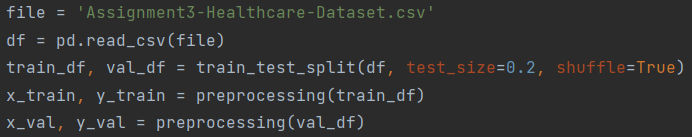
The first thing to do is to define an objective function to pass it to the study.



The objective function is where we initialize the LightGBM model, train it, and then return an evaluation of how well the model did (in this case is the f1 macro score). Here’s the full function:

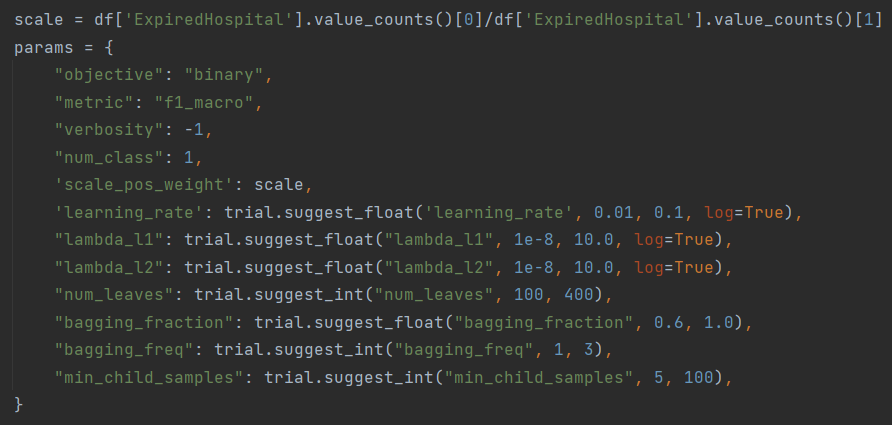


Now I’ll go through each segment of the function:



Preprocessing the data to train the model.

The parameter distributions of the LightGBM model.



Passing the params to the model.



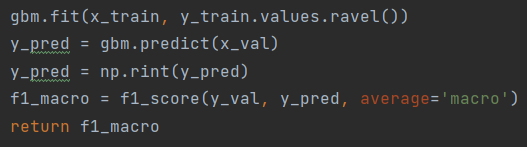
Here are the explanations for some of the main parameters:

* + *"objective": "binary"* – because we’re doing binary classification.
  + *"metric": "f1\_macro"* – to match the scoring of the specification.
  + *"scale\_pos\_weight":* *scale* – to address the unbalanced dataset.

The parameters with *trial.suggest\_float()/suggest\_int()* are the ones that will be tuned during the study.

* + *"learning\_rate"* – the learning rate of the model
  + *"num\_leaves"* – the maximum number of leaves in one tree.
  + *"bagging\_fraction"* – only select this fraction of the data, used deal with overfitting.
  + *"bagging\_freq"* – determines how frequent the bagging will happen.
  + *"min\_child\_samples"* – the minimum number of data in one leaf, used to deal with overfitting.

After creating the model and choosing a set of parameters, the function will evaluate the model and return the f1 macro score.



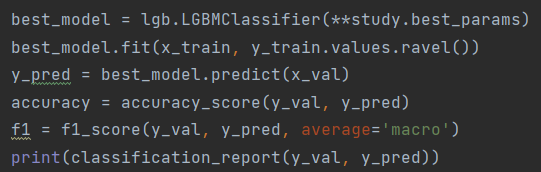
The Optuna study will use this model’s score to optimize on the next model. Each trial will look like this:

[I 2023-10-31 22:55:31,574] Trial 0 finished with value: 0.7610783001521474 and parameters: {'learning\_rate': 0.01054382280983923, 'num\_leaves': 367, 'bagging\_fraction': 0.9887261328007981, 'bagging\_freq': 2, 'min\_child\_samples': 40}. Best is trial 0 with value: 0.7610783001521474.

[I 2023-10-31 22:55:32,600] Trial 1 finished with value: 0.7776273352363949 and parameters: {'learning\_rate': 0.021950396513237273, 'num\_leaves': 145, 'bagging\_fraction': 0.6513442892648814, 'bagging\_freq': 3, 'min\_child\_samples': 28}. Best is trial 1 with value: 0.7776273352363949.

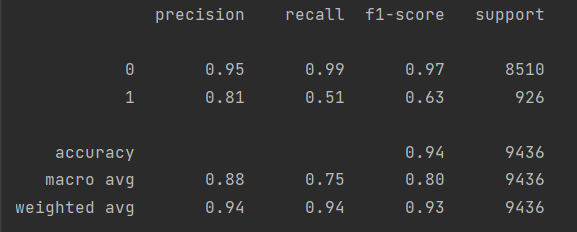
[I 2023-10-31 22:55:33,764] Trial 2 finished with value: 0.7915252891017404 and parameters: {'learning\_rate': 0.06845566819754405, 'num\_leaves': 198, 'bagging\_fraction': 0.9017391203382289, 'bagging\_freq': 1, 'min\_child\_samples': 72}. Best is trial 2 with value: 0.7915252891017404.

After the study has gone through all of the trials, the model with the best parameters will be returned.



A new LightGBM model will be created using the best parameters that the study found, and then this model will be evaluated.

1. Results evaluation:

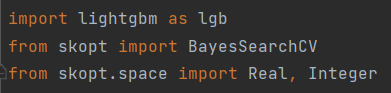


This is the best result that I could get from using Optuna, the f1 macro score ranges between 0.77 and 0.80, which is not bad. But worse than the Random Forest classifier and LightGBM with **BayesSearchCV**, which I’ll be covering next.

**LightGBM with BayesSearchCV:**

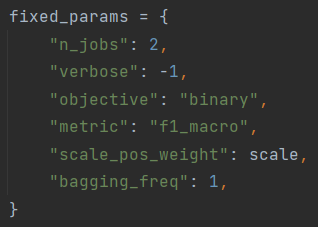
1. Initialization:

Import **BayesSearchCV**from the *skopt* packageand *lightgbm:*



Import **Real, Integer** from *skopt.space* so we can later define the parameter distributions for the hyperparameters tuning process.

Initializing the **LGBMClassifier()** with some fixed parameters.

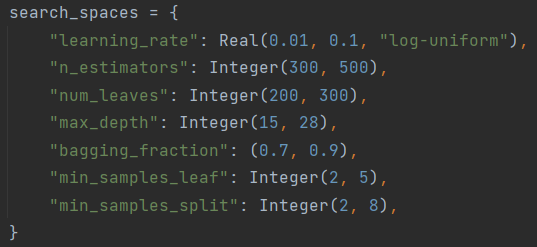




* + *"objective": "binary"* – because we’re doing binary classification.
  + *"metric": "f1\_macro"* – to match the scoring of the specification.
  + *"scale\_pos\_weight":* *scale* – to address the unbalanced dataset.
  + *“bagging\_freq”: 1* – Apply bagging on all iterations, deals with overfitting.

1. Hyperparameters tunning:

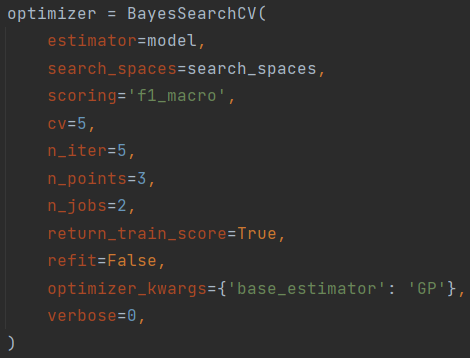
First, we create the parameter distributions for all of the parameters we want to tune.



An explanation on these parameters:

* + *"learning\_rate"* – the learning rate of the model
  + *"n\_estimators"* – the number of trees used to make the model.
  + *"num\_leaves"* – the maximum number of leaves in one tree.
  + *"bagging\_fraction"* – only select this fraction of the data, used deal with overfitting.
  + *"bagging\_freq"* – determines how frequent the bagging will happen.
  + *"min\_child\_samples"* – the minimum number of data in one leaf, used to deal with overfitting.
  + *"reg\_alpha"* –L1 regularization, helps generalize the model.
  + *"reg\_lambda"* –L2 regularization, also helps generalize the model.

After that, we define the **BayesSearchCV**:



The **BayesSearchCV** will use the model and the parameter distributions that were created prior.

The metric score will be “f1\_macro”, and the number of folds will be 5.

It is also using the Gaussian Process (GP) as part of the optimization process. Therefore, it’s possible to use a lower number of iterations compared to **RandomizedSearchCV.**

Now we can start the tuning process.



After the optimizer has finished tuning, the best parameters will be returned using this line of code.

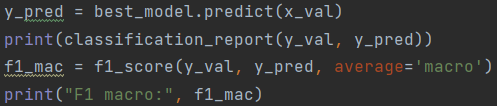


We then create a new **LGBMClassifier** using the best hyperparameters after tuning, along with the fixed parameters.

A black screen with white text

Description automatically generated

Now this model will be evaluated.



1. Results evaluation:

A screenshot of a computer

Description automatically generated

This model got a 0.823 in terms of F1 macro, which is the highest score of all my models.

Note that this is not the best LGBM model that I’ve trained. The best that I’ve gotten scored a 0.825 in practice and a 0.827 on Kaggle. But I unfortunately didn’t save it.



And this is my best Kaggle submission till now.

Here are some other submissions using this classifier:



1. **The best classifier and how it works:**

Here’s a table detailing the results of all the classifiers that I used.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Classifier/Score | F1 macro | Accuracy | Precision | Recall |
| LinearSVC | 0.71 | 0.92 | 0.86 | 0.66 |
| KNN | 0.71 | 0.92 | 0.78 | 0.67 |
| Neural Network | 0.73 | 0.85 | 0.69 | 0.85 |
| Random Forest | 0.80 | 0.93 | 0.84 | 0.77 |
| XGBoost | 0.81 | 0.94 | 0.87 | 0.77 |
| LGBM with **Optuna** | 0.80 | 0.94 | 0.88 | 0.75 |
| LGBM with **BayesSearchCV** | 0.82 | 0.94 | 0.84 | 0.81 |

As can be seen, if we are scoring just based on F1 macro or Recall, the LightGBM model with **BayesSearchCV** is the best model.

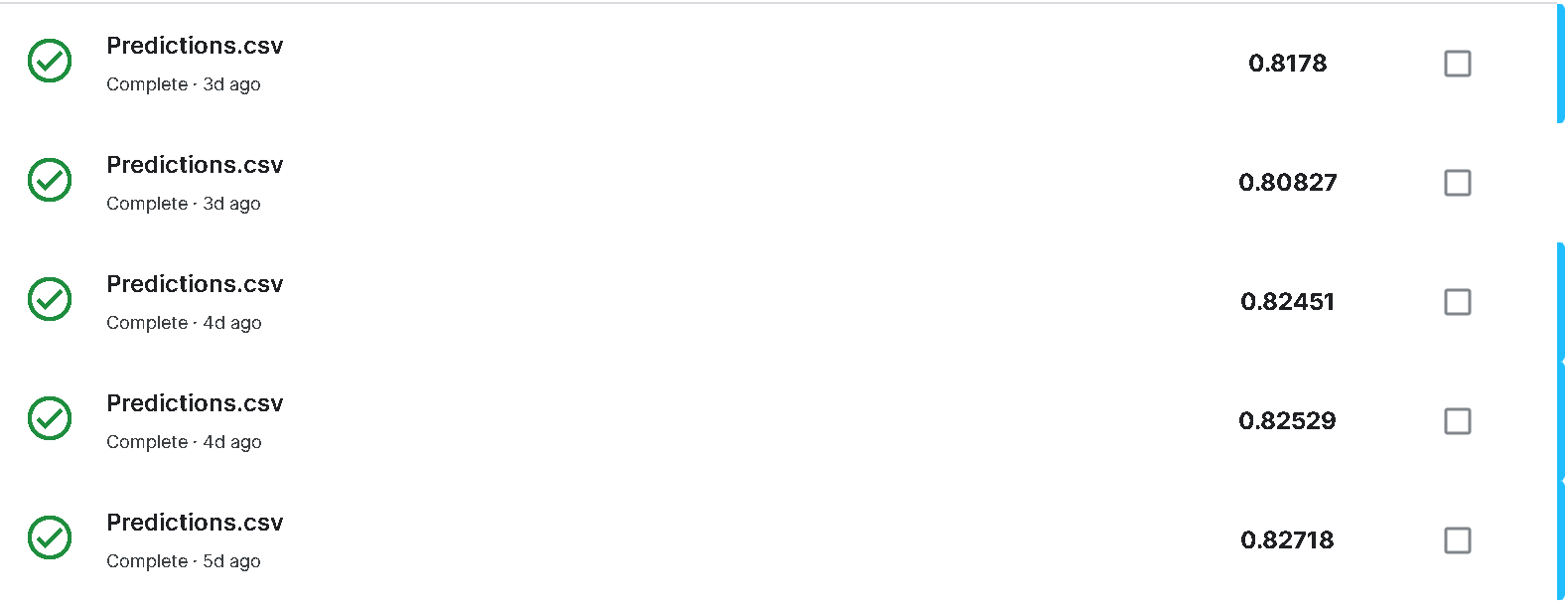
If we are scoring on accuracy then it’s a close call between XGBoost and LGBM with either **Optuna** or **BayesSearchCV**.

If we are going by Precision score alone then LGBM with **Optuna** did the best in this circumstance.

So out of all my classifiers, the best one is definitely LightGBM.

Here’s a quick rundown of what is LightGBM:

* **Type:** LightGBM is a Gradiant Boosting Tree framework.
* **Performance:** Even though I didn’t save the models that did well, I did submit all of them to Kaggle. So here are the scores these LGBM models achieved on Kaggle:

****

****

And 0.827 was the highest that this classifier got, and that is also my top score.

* **How the classifier solved the problem:**

Gradient Boosting Tree (GBT) is a machine learning algorithm that combines multiple decision trees to create a more accurate model.

Here’s an overview of how it works:

1. Trains a tree.
2. Calculate the residuals (the difference between the predicted and actual values).
3. Using Gradient Descent, a new tree will be trained on the residuals in order to minimize the loss.
4. The new residuals are updated by adding the predictions of the new tree to the previous predictions.
5. Steps 3-4 are repeated until a stopping criterion is met.

The LightGBM framework added Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB) to the GBT algorithm. I won’t go into details of what these two ideas are, but they essentially make the GBT classifier a lot more efficient.

* **Reasons of choosing this classifier:**
  + The data has both type of values: numerical and categorical. And LightGBM can handle both of them.
  + The dataset also has some features with a lot of missing values, which LightGBM without needing to fill those missing values.

1. **Kaggle Submission Score:**