The problem: - COVID-19 Outcome Prediction

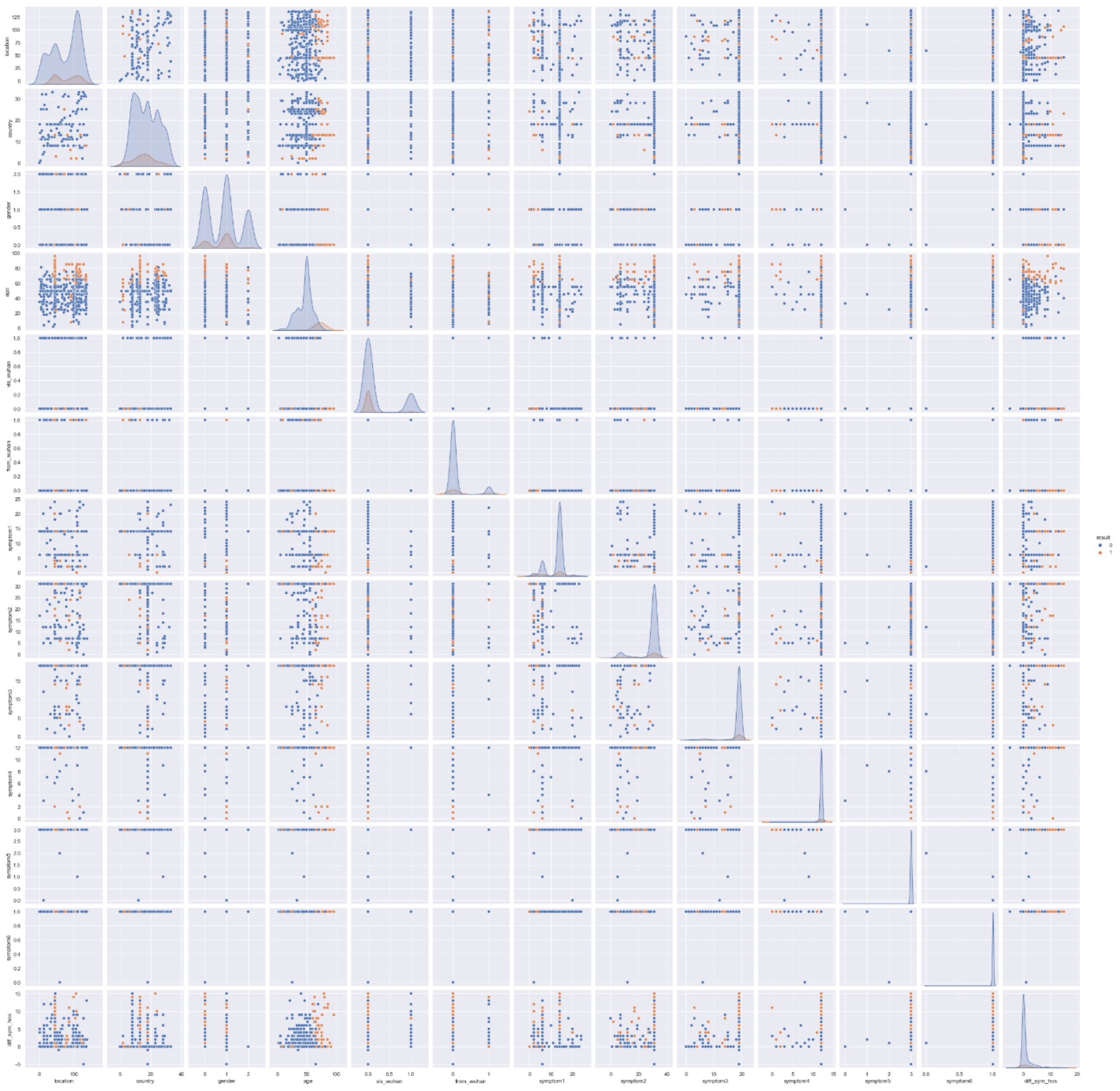
The data used in this project will help to identify whether a person is going to recover from coronavirus symptoms or not based on some pre-defined standard symptoms.

The data

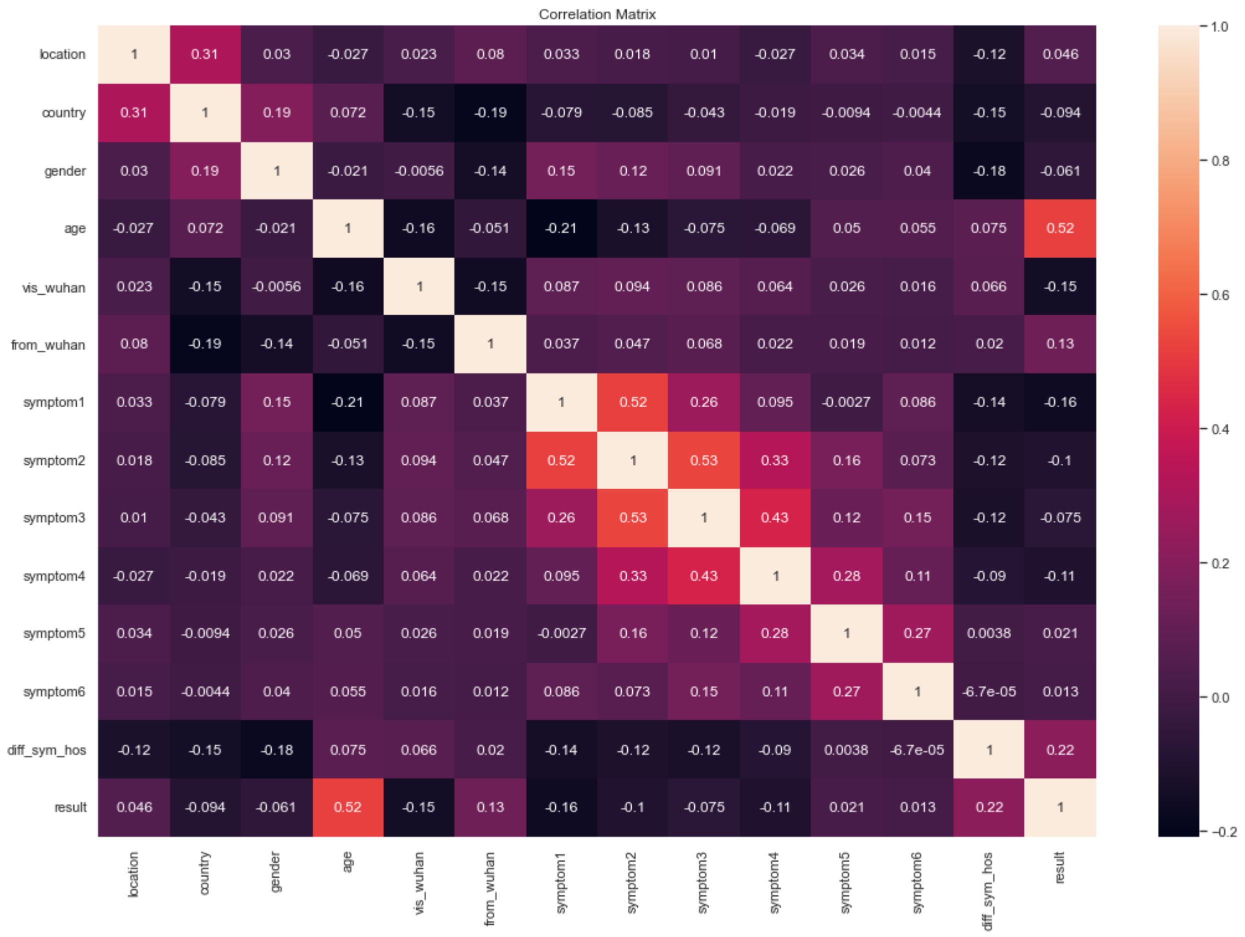
the first step in our project is loading the dataset (“data.csv”) and describing it. After that step we notice that the data set contain many features ('country','location','age','gender','visited_Wuhan','From_Wuhan','age','symptom1','symptom2','symptom3','symptom4') and class label ('result'). The data set has 863 records and 14 features. The problem in that data set: - the number of class (0) is 755 and the number value of class (1) is 108 which mean the class label is highly unbalanced it will result the splitting to be unfair for both classes. We can solve it by weighting the model (increasing class 1) and splitting and stratifying the data evenly between both labels. We Weighted the models (Weighted Logistic Regression - Naive Bayes priors) in our project. The two models’ classes provide the class weight argument that can be specified as a model hyperparameter. The class weight is a dictionary that defines each class label (0 and 1) and the weighting to apply in the calculation of the negative log-likelihood when fitting the model.

Visualize the data to gain insights

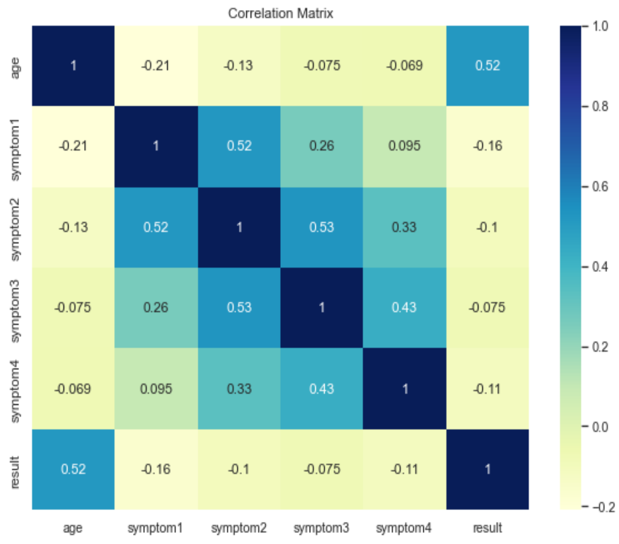
we provide some insights and visualization that would help us understand the data and what we should expect from our models.



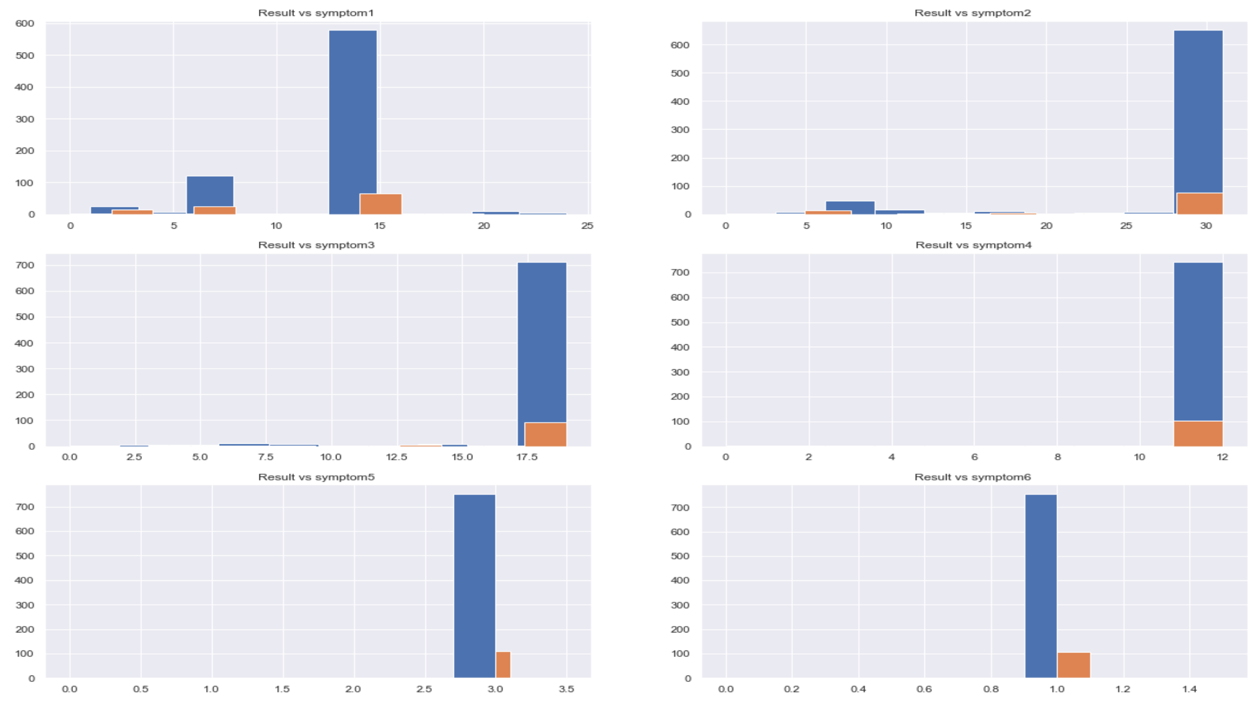
from that visualization the labels can hardly be separable (both labels share a lot of values) in all features except for ("age") we need to further explore the correlation



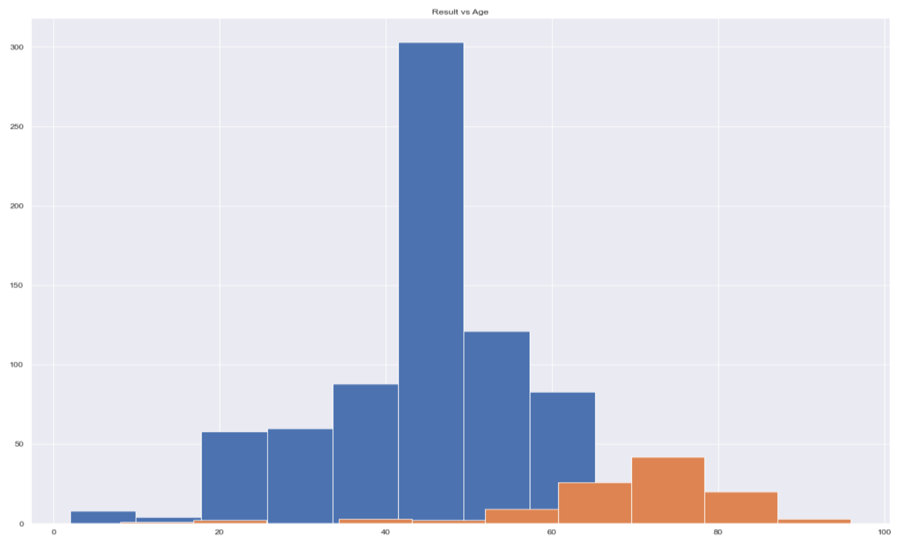
from that correlation matrix, we notice that the features ("Age" and "diff_sym_hos") have the highest correlation with the results.



from these figures, we notice that these features (age and symptoms) need further exploration and we found interesting results.



First:- We can see that the values for both recovered and dead patients are concentrated at certain values in all symptoms almost identically associated.
Second:- all symptoms peak at a certain value, which means it’s a recurrent symptom that we should emphasize by feeding these values as categories and ignoring the order of the value.



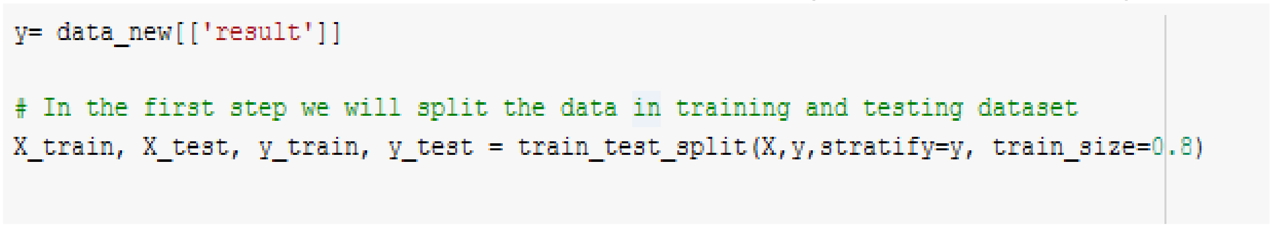
from that visualization between age and result, we notice that 'Age' is a key feature for predicting the results with only a few overlaps between ages from 40-60

Handling categorical data by Applying one-hot encoding

we need to ignore the order of the values in symptoms because they are categorical, a symptom with value 9 is not more serious than a symptom with value 8 also the locations are not ordinal data so we will hot encode it only 'age' and "time before symptoms" are ordinal data
the rest should be encoded.

Train – test split

The train test split is one of the hardest hyperparameters to configure especially for small, unbalanced data set like this.
First, we need to ensure that we are splitting the correct train-split ratio, so we ran the model multiple times to find the best ratio.
Second, because we have a higher number of recovered patients than dead patients, we need to ensure that the train and test datasets are representative of the original

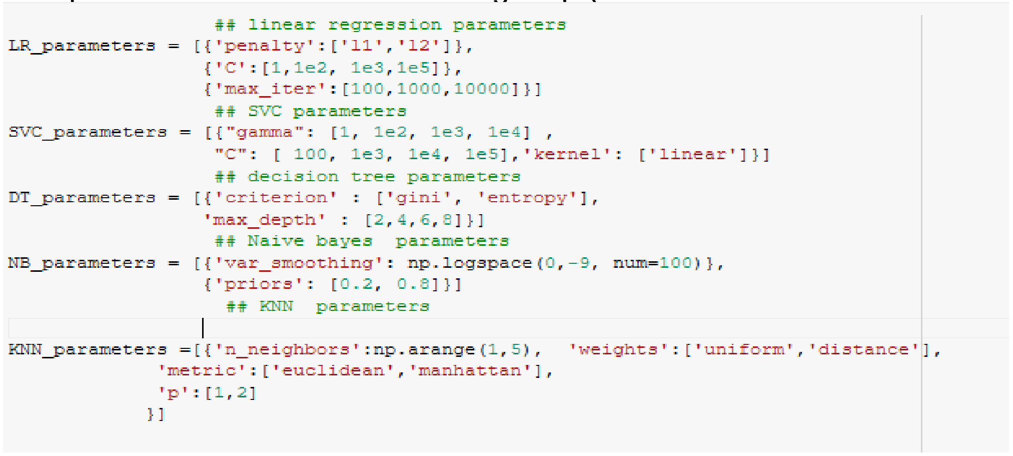


```
y= data_new[['result']]

# In the first step we will split the data in training and testing dataset
X_train, X_test, y_train, y_test = train_test_split(X,y,stratify=y, train_size=0.8)
```

dataset.
We used “stratify” to ensure that the same ratio is taken from each class

Model building and tuning

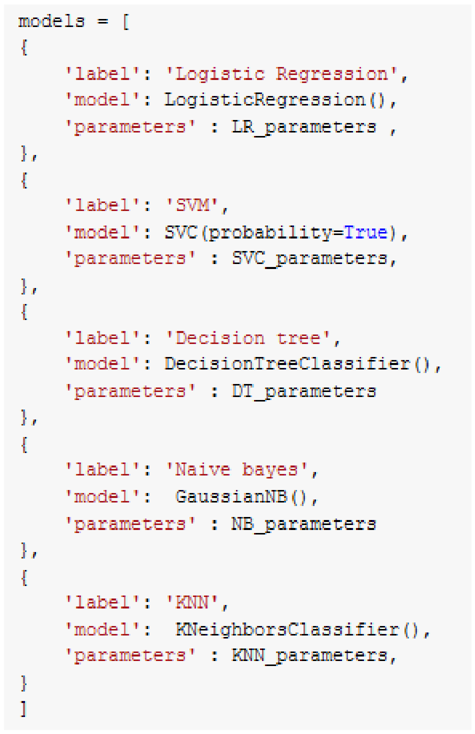
We have 5 models with multiple hyperparameters and multiple implementation methods we needed a quick way to test our models and find the best parameters so we can try different parameters outside the training loop (different metrics and train–test ratio)



```
                    ## linear regression parameters
LR_parameters = [{'penalty':['l1','l2']},
                 {'C':[1,1e2, 1e3,1e5]},
                 {'max_iter':[100,1000,10000]}]
                    ## SVC parameters
SVC_parameters = [{"gamma": [1, 1e2, 1e3, 1e4] ,
                  "C": [ 100, 1e3, 1e4, 1e5],'kernel': ['linear']}]
                    ## decision tree parameters
DT_parameters = [{'criterion' : ['gini', 'entropy'],
                 'max_depth' : [2,4,6,8]}]
                    ## Naive bayes  parameters
NB_parameters = [{'var_smoothing': np.logspace(0,-9, num=100)},
                 {'priors': [0.2, 0.8]}]
                     ## KNN  parameters

KNN_parameters =[{'n_neighbors':np.arange(1,5),  'weights':['uniform','distance'],
             'metric':['euclidean','manhattan'],
             'p':[1,2]
            }]
```

First, we defined all the parameters that we want to test initially, each parameter is a list of dictionary of the values that the grid search will go through to find the optimal combination.

Next, we defined our model list to make it easier to go through different implementations in learning:

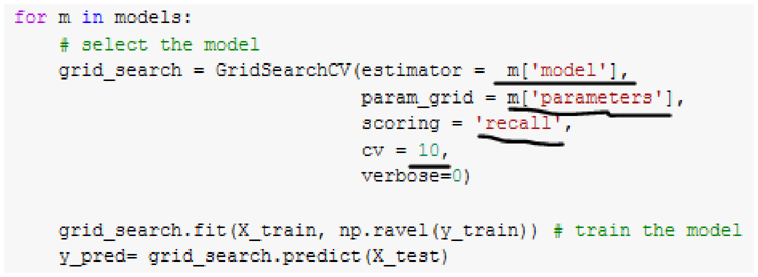


```
models = [
{
    'label': 'Logistic Regression',
    'model': LogisticRegression(),
    'parameters' : LR_parameters ,
},
{
    'label': 'SVM',
    'model': SVC(probability=True),
    'parameters' : SVC_parameters,
},
{
    'label': 'Decision tree',
    'model': DecisionTreeClassifier(),
    'parameters' : DT_parameters
},
{
    'label': 'Naive bayes',
    'model':  GaussianNB(),
    'parameters' : NB_parameters
},
{
    'label': 'KNN',
    'model':  KNeighborsClassifier(),
    'parameters' : KNN_parameters,
}
]
```

Each model has a corresponding parameter that the training loop is going to take.
For the logistic regression we used :
1- logistic Regression() that includes penalty(L1&l2)
2- SVM: linear SVC () with different gammas and C
3- decision tree classifier () with (Gini entropy), and multiple depths to prevent overfitting.
4- naïve Bayes() with smoothing because we would have a lot of 0 values, and priors to weigh the unbalanced class
5- KNN classifier, with NN from (1to5) we don't want the model to overfit, different distance measurements and metrics.

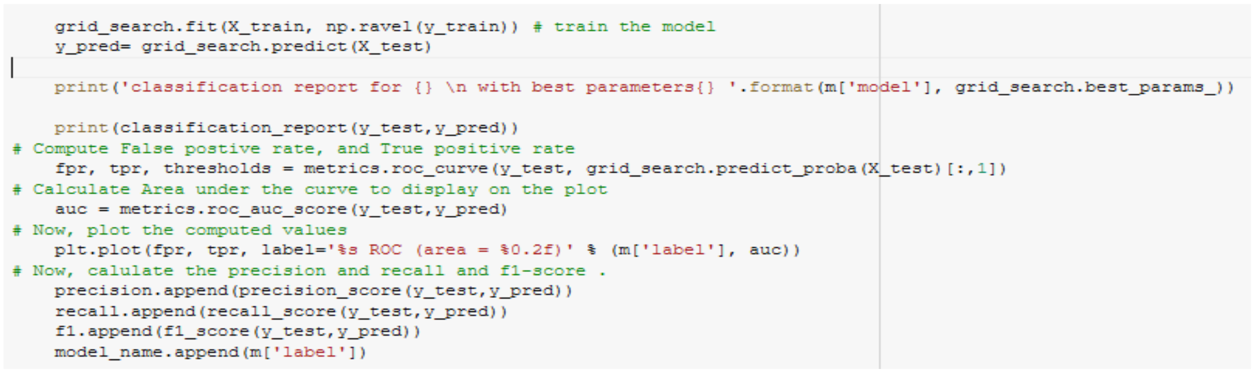
The training loop

Our goal is to find the best hyperparameters and print the result of all the models, so it is inconvenient to train and tune each model separately, although this training loop is computationally heavy it saved us a lot of time to find the values that we want to change We used grid search as our estimator, fed the parameters as a list of parameters in each model, and 10 cross-validation epochs to find the best value (later we reduced that after we found the optimal parameters)



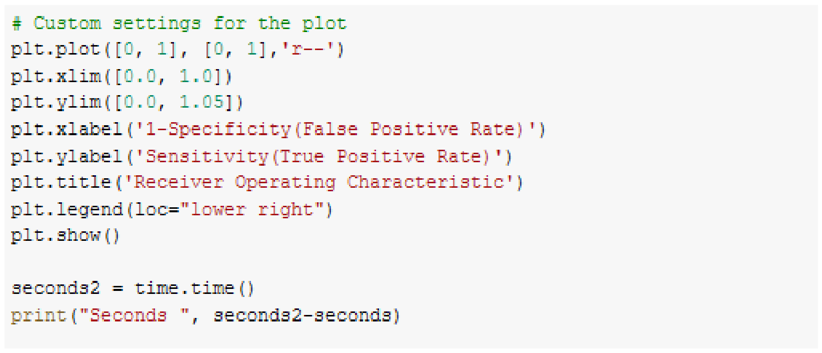
```
for m in models:
    # select the model
    grid_search = GridSearchCV(estimator =  m['model'],
                               param_grid = m['parameters'],
                               scoring = 'recall',
                               cv = 10,
                               verbose=0)

    grid_search.fit(X_train, np.ravel(y_train)) # train the model
    y_pred= grid_search.predict(X_test)
```

Then we fit the model, predict with the best parameter then append the best result that will later be used to create a data frame for all the results to compare each training run and select the best.

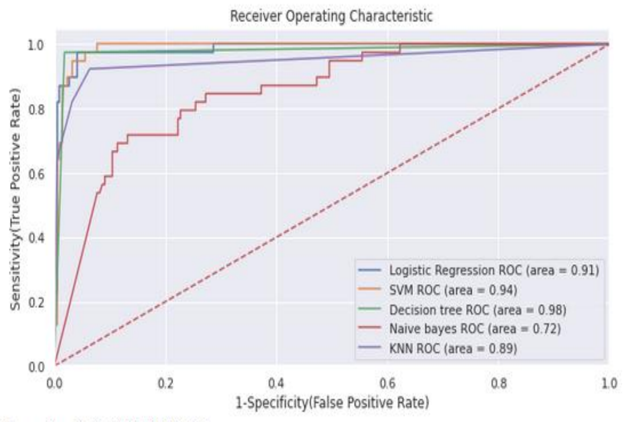


```
    grid_search.fit(X_train, np.ravel(y_train)) # train the model
    y_pred= grid_search.predict(X_test)

    print('classification report for {} \n with best parameters{} '.format(m['model'], grid_search.best_params_))

    print(classification_report(y_test,y_pred))
# Compute False postive rate, and True positive rate
    fpr, tpr, thresholds = metrics.roc_curve(y_test, grid_search.predict_proba(X_test)[:,1])
# Calculate Area under the curve to display on the plot
    auc = metrics.roc_auc_score(y_test,y_pred)
# Now, plot the computed values
    plt.plot(fpr, tpr, label='%s ROC (area = %0.2f)' % (m['label'], auc))
# Now, calulate the precision and recall and f1-score .
    precision.append(precision_score(y_test,y_pred))
    recall.append(recall_score(y_test,y_pred))
    f1.append(f1_score(y_test,y_pred))
    model_name.append(m['label'])
```

And finally, we plot the ROC, AUC of all the models in one graph and compute running time:



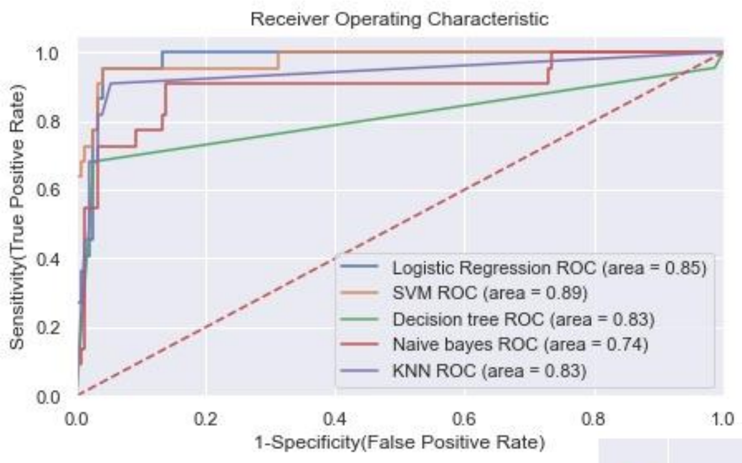
```
# Custom settings for the plot
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('1-Specificity(False Positive Rate)')
plt.ylabel('Sensitivity(True Positive Rate)')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

seconds2 = time.time()
print("Seconds ", seconds2-seconds)
```

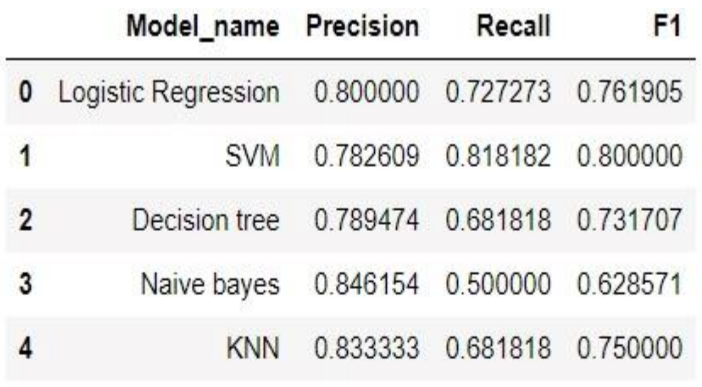


Comparative analysis:

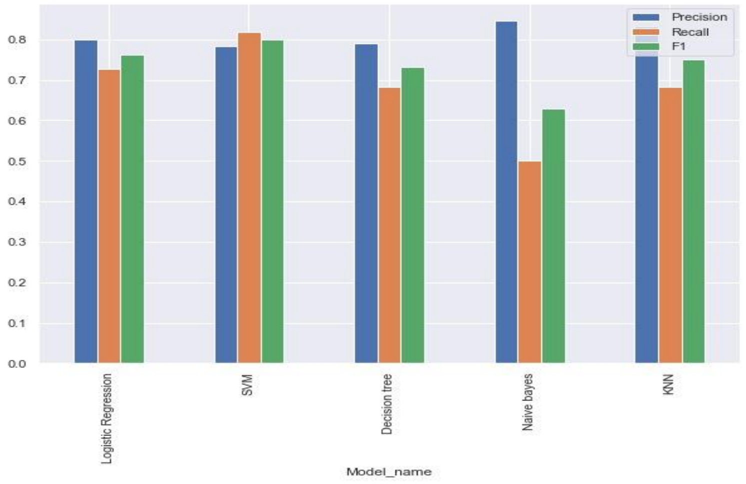
We need to compare our results for different hyperparameters, train-test ratio, and concerning different metrics (accuracy and recall)
to save us time and effort we used grid search to find the optimal parameters
we used an All-in-one-for-loop to train the 5 models with grid search and we noticed a lot of interesting results:
All the models had acceptable results in recall and accuracy except for naïve Bayes.

train-test ratio: 0.8 , metric : accuracy.

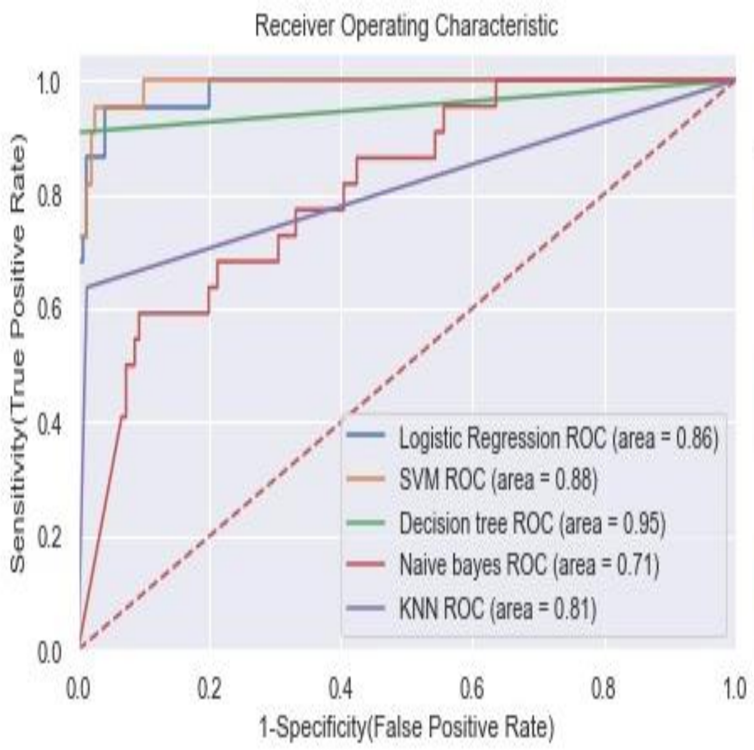


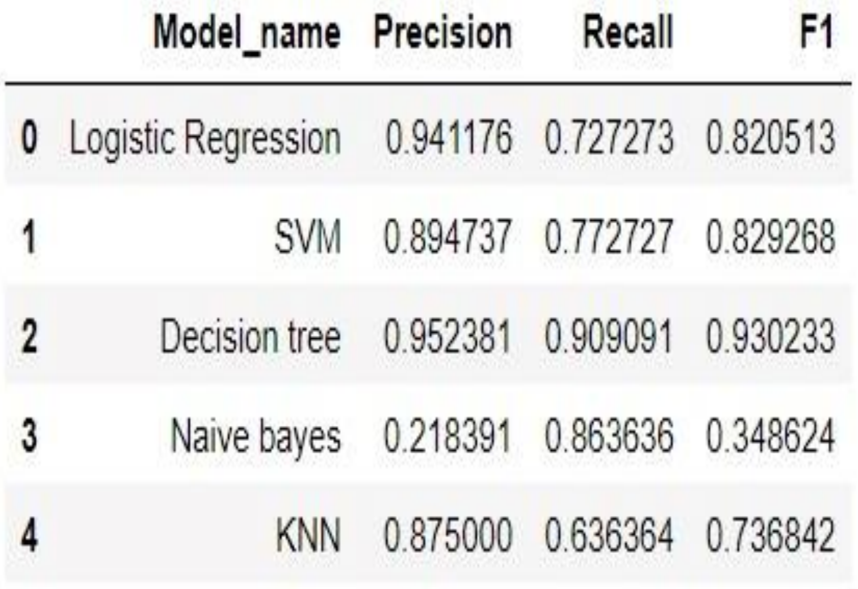


	Model_name	Precision	Recall	F1
0	Logistic Regression	0.800000	0.727273	0.761905
1	SVM	0.782609	0.818182	0.800000
2	Decision tree	0.789474	0.681818	0.731707
3	Naive bayes	0.846154	0.500000	0.628571
4	KNN	0.833333	0.681818	0.750000

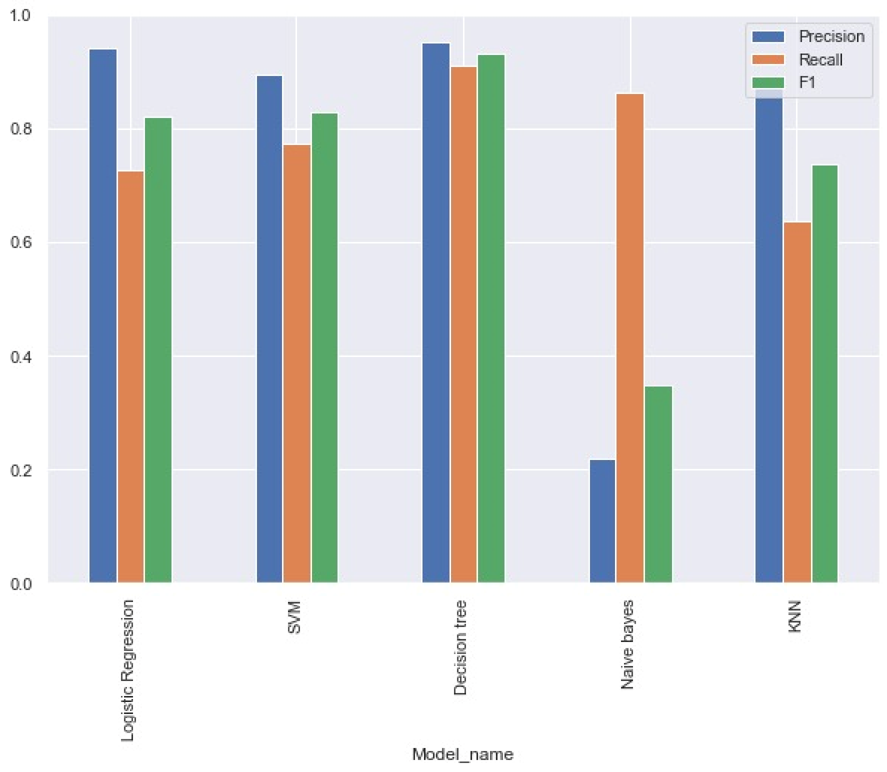


train-test ratio: 0.8, metric: recall, with stratify.

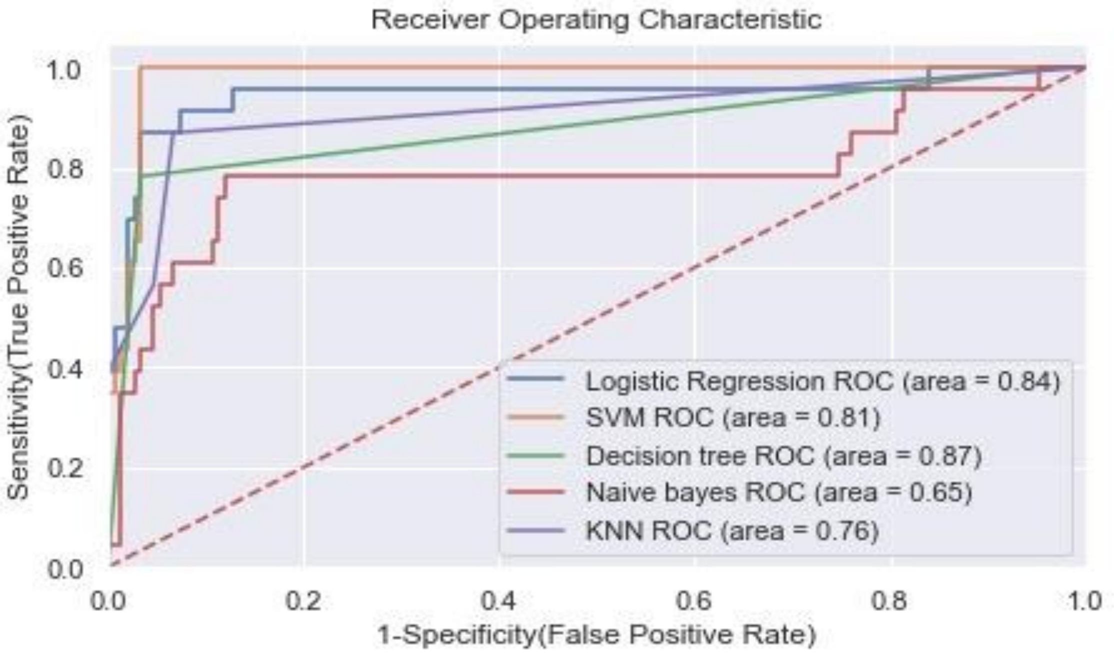


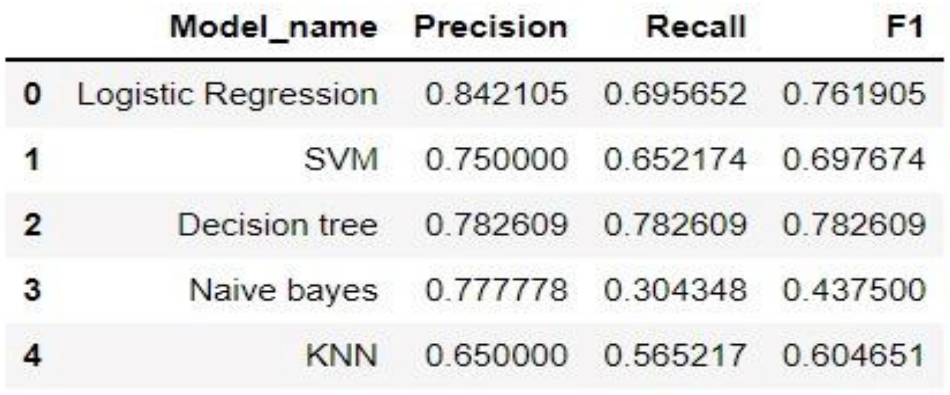


	Model_name	Precision	Recall	F1
0	Logistic Regression	0.941176	0.727273	0.820513
1	SVM	0.894737	0.772727	0.829268
2	Decision tree	0.952381	0.909091	0.930233
3	Naive bayes	0.218391	0.863636	0.348624
4	KNN	0.875000	0.636364	0.736842

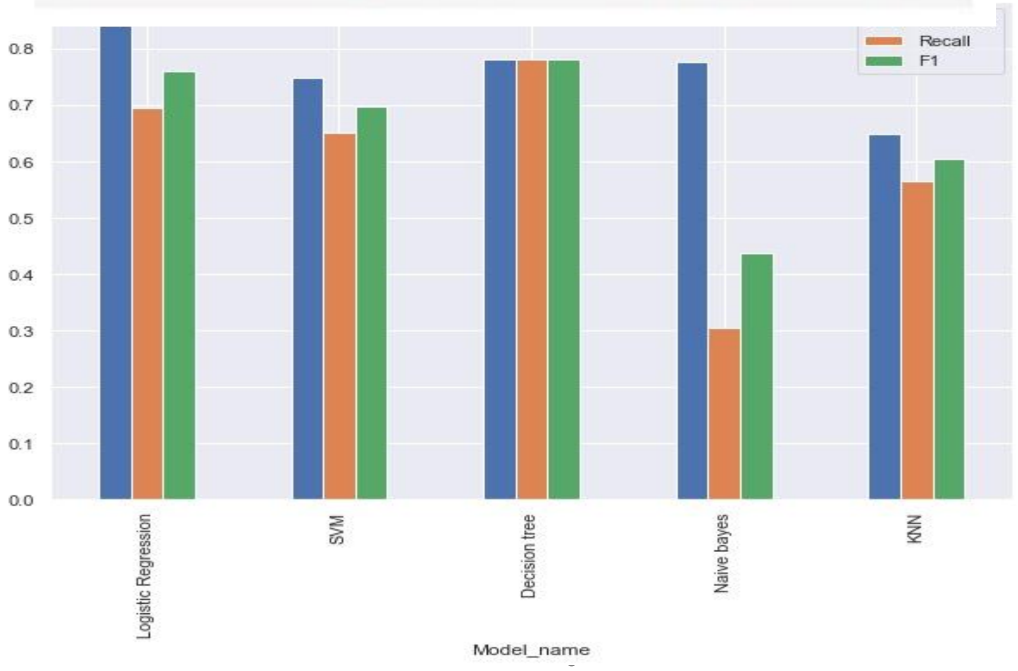


train-test ratio: 0.8, metric: recall, without stratify.

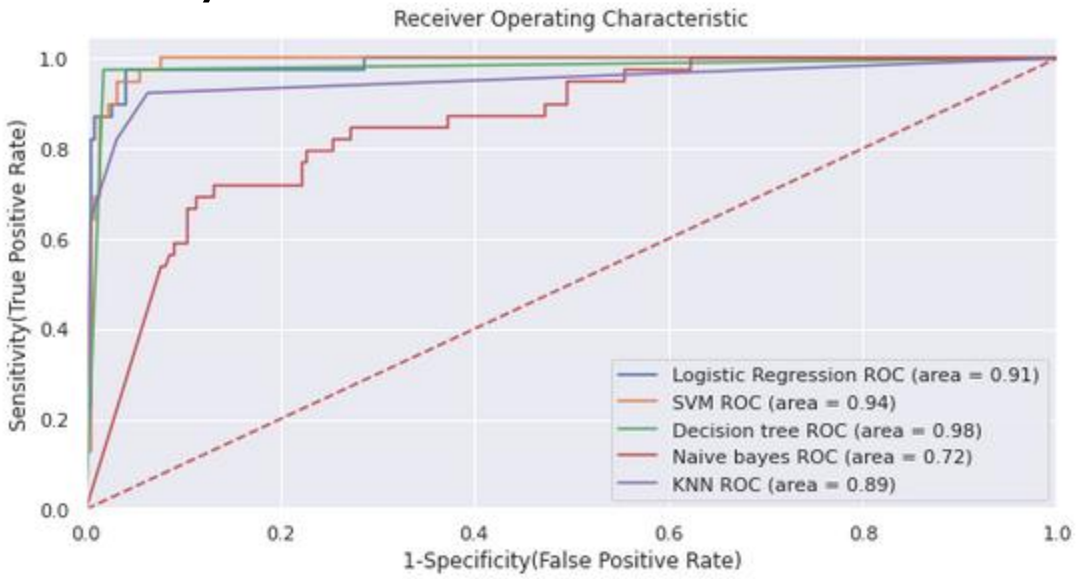


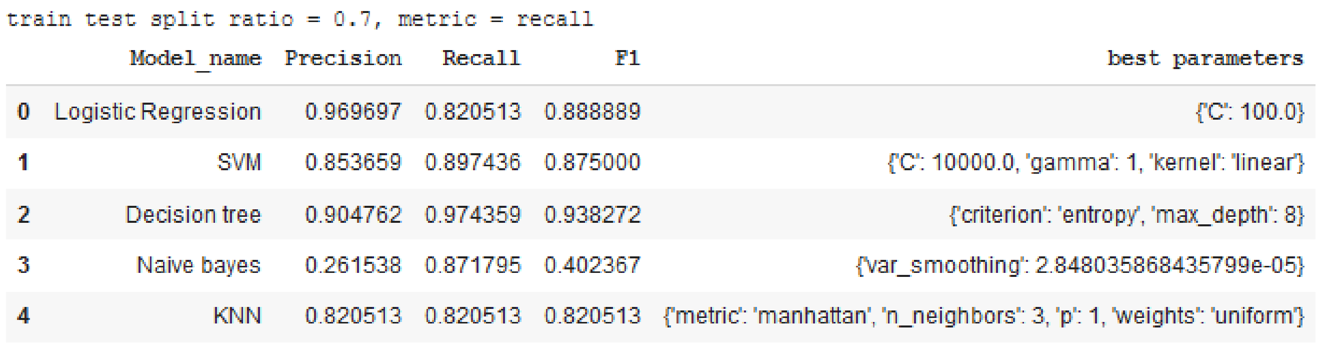


	Model_name	Precision	Recall	F1
0	Logistic Regression	0.842105	0.695652	0.761905
1	SVM	0.750000	0.652174	0.697674
2	Decision tree	0.782609	0.782609	0.782609
3	Naive bayes	0.777778	0.304348	0.437500
4	KNN	0.650000	0.565217	0.604651



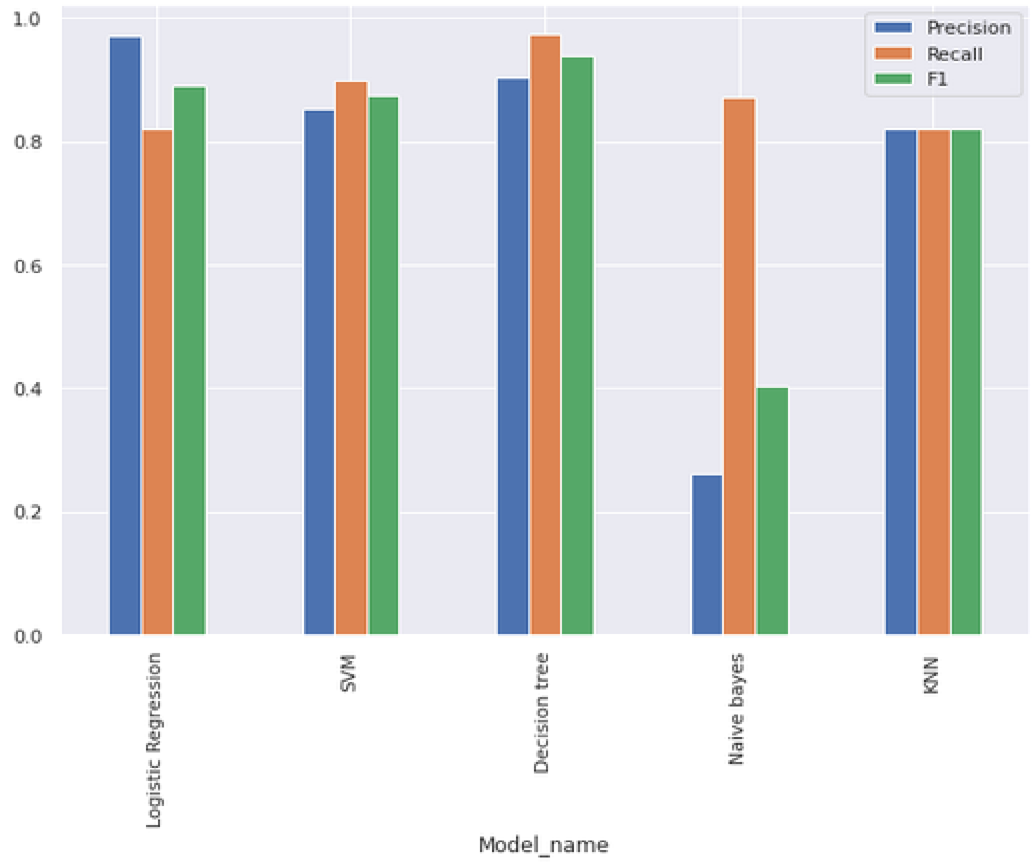
train-test ratio: 0.7, metric: recall, with stratify (best parameters).





```
train test split ratio = 0.7, metric = recall
```

	Model_name	Precision	Recall	F1	best parameters
0	Logistic Regression	0.969697	0.820513	0.888889	{'C': 100.0}
1	SVM	0.853659	0.897436	0.875000	{'C': 10000.0, 'gamma': 1, 'kernel': 'linear'}
2	Decision tree	0.904762	0.974359	0.938272	{'criterion': 'entropy', 'max_depth': 8}
3	Naive bayes	0.261538	0.871795	0.402367	{'var_smoothing': 2.848035868435799e-05}
4	KNN	0.820513	0.820513	0.820513	{'metric': 'manhattan', 'n_neighbors': 3, 'p': 1, 'weights': 'uniform'}



Deductive reasoning:

From these results, we conclude that the best train-test ratio is 0.7, and stratifying the labels had a significant effect on the model performance. These are only the best results that we had, we also tried other train-test ratios but anything greater than 0.8 or less than 0.7 had significantly less recall so we discard them from our report.
As for the parameters, it would take us no less than 10 pages to discuss the work of 3 individuals for 4 straight days but the parameters that significantly affected the result was:

1- Stratify=y for the train-test
2- Metric=recall for the grid search
3- Kernel = linear for SVM
4- Var smoothing for naïve bayes
5- Max depth = 8 for decision trees
6- Neighbors = 3
7- Penalty = l1 for logistic regression
8- Weighted class for logistic regression

conclusion:

after multiple observations, we conclude that it would be most suitable to use decision trees for the problem at hand, not only It achieves the best scores it would be easy for doctors to make decisions based on it and it will be easier to grasp without a technical background.

The second-best model is SVM because we are dealing with high-dimensional data.