ARESD Report

Data Science Research Workshop

NadirJalaludeen

alan-man

Sinan

Summary

Introduction

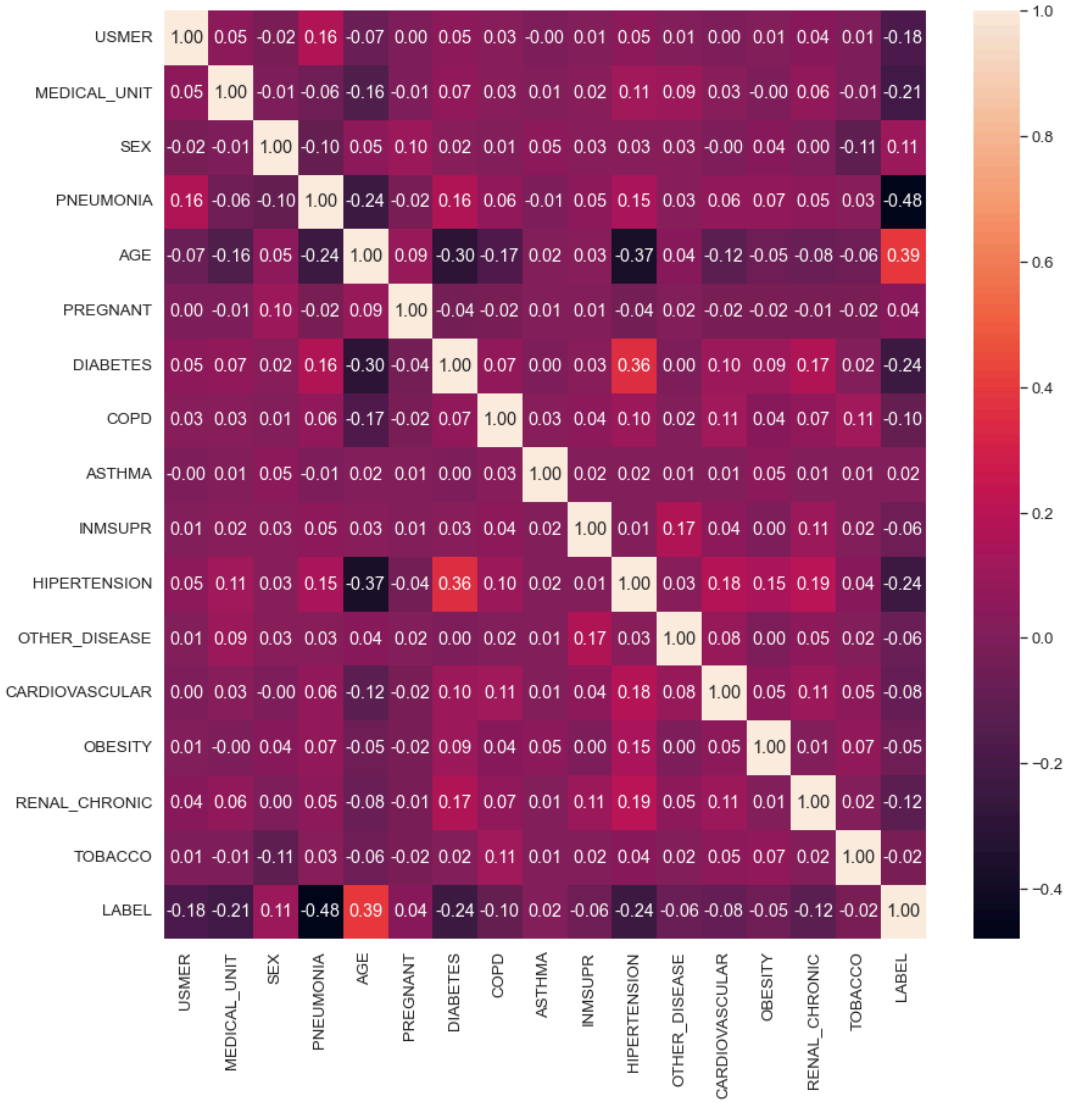
The usage of data science during the Covid-19 pandemic has been really important in almost every domain (searching for a cure, assisting healthcare personnel, controlling the population, etc...)[1]. In order to best assist the different patients, we have to determine if they are at high risk or not.
The goal of this project is to build a machine-learning model that, given a Covid-19 patient's current symptoms, condition, and medical history, will predict whether the patient is at high risk or not. The variable to be predicted is the variable LABEL which measures the severity of the COVID-19 infection. This project is part of a Kaggle competition, on which the final predictions should be submitted.[2]

I - Description of the dataset

1 - Analysis of the dataset

On the Kaggle page of the competition, we can find 3 different .csv files.
First, we have a training dataset (train.csv) containing 30051 individuals infected with the virus. In this file, there are 17 columns in total, with the 16 first being features like age, sex, and many medical conditions such as asthma, diabetes, obesity, and so on. The last column represents the Label with the different outcomes for each individual.
Then, we have a testing dataset (prev.csv) containing information about 7512 patients that we need to classify.
The main goal is to predict as many correct predictions as possible using the first 16 columns.

2 - Data Preprocessing



The first thing that we did was plotting this correlation heatmap with the seaborn library[3] (Cf. appendix 1). We can see that some features have a higher correlation than others. So, it could be relevant to only build our model by using only a few of them. When we tested the different models, in order to determine the accuracy, we tested them on two different datasets. The first was composed of every feature, and the second one only contained the features that have a correlation greater than 0.1 or lesser than -0.1, representing 8 of the 16 total features.

II - All the different ML models implemented

1 - Accuracy estimation methodology

To estimate the accuracy of our models, we first divided our train dataset into a test dataset and a training dataset randomly by using the split function (Cf. appendix 2).
At each time that we used this function, we set a random seed equal to 42 in order to have a more accurate comparison.
We trained our model by using the training dataset, and we measured its accuracy by using the test function.

2 - K-Nearest Neighbors

The K-Nearest Neighbors algorithm seems extremely simple but is quite efficient for classification. The basis of the logic is to classify a new subject into a predefined category.
The algorithm has 3 main parts: distance metric, k value, and classification. KNN is a distance-based classifier, meaning that it assumes that the smaller the distance between two points, the more similar they are. To calculate the distance we can use the Euclidean Distance, which is a more specific use of the Minkowski Distance.
Then, the K value is simply the number of values considered. Finally, to classify we can average the k closest values or majority.
We observe that variance increases when we take a small k and the bias increases with a bigger k. By testing this classifier with different k-values and different split random seeds, we figured out that 21 is the best k value.
Although the algorithm is simple it can be quite heavy in computing and memory usage.
In the training phase, it stores all the training data in memory, and in the testing phase, it searches for its neighbors to the new observation using the distance metric. If the dataset is too big or there are too many features, then the program can be inefficient which is certainly our case. With every feature, we had a precision of 64,28%, and with the selection of features, we had a precision of 64,6%. In this case, preprocessing was useful.

3 - Naive Bayes and Weighted Naive Bayes

The next model that we tested was a good old Naive Bayes classifier which thrives on modeling classes or categories, but also with high-dimensional data (with many features), which fits our case with covid patients because we have 16 features and the outcome we try to predict is categorical: Soft Covid, Strong Covid or Dead.
However this model is based on a huge simplifying assumption which is ignoring the interdependency of the features, you might think this is not suitable for covid because obviously some features are related such as age and some medical conditions, and you would be right. This is why this classifier didn't get us the best results.
This model gave us a precision of 65,11% with every feature, and 65,38% with the selection of features. It's also an improvement here, since we decrease the naivety of our model.

But we also tried to add some weights to certain features by adding an exponent to the formula.
Therefore, P(A|B) = P(B|A) * P(A) / P(B) became P(A|B) = P(B|A)^w * P(A) / P(B).
We found an estimation of the best weight by coding an exhaustive research algorithm that looks for the best accuracy on himself (the training data), while varying his weights. But finding the best weights with this method can take a really long time, eventually an hour. This Naive Bayes is not so naive anymore.
We had a precision of 67,29% without preprocessing, and 66,88% with. Here, the preprocessing was not a good idea. It's easily explained by the fact that, when a certain feature is almost not correlated, the algorithm will find a little exponent by itself, while keeping the feature that could have a small impact.

4 - Decision Tree

The third model that we implemented was the decision tree, and specifically classification tree since we have to predict categories. It's an accurate algorithm for classification as the image of a tree-like model is a great way to visualize why a certain outcome is likelier than another.
The method is simple: we recursively split the training data into subsets based on the splitting criteria (lowest Gini impurity or entropy, information gain ...) until a stopping variable such as maximum depth to prevent overfitting and making the tree too complicated. Like with KNN, the more the depth increases, the more the variance increases. And the more the depth decreases, the more the bias increases.
Then in the testing phase, the algorithm follows the tree to predict the label.
In our case, we found the best depth to be 7. With this hyperparameter, we had a precision of 67,6% without processing, and 67,48% with.

5 - Voting

Now, having implemented multiple models, why not aggregate them and get an even better prediction, sounds good huh? Well it's not that simple. We combined our bayes classifier, the KNN model and the regression tree.
Our voting is literally like a vote, for each patient, if two of the models foresee the same outcome then the voting will predict that label even if the third model disagrees.
At the end of the day, this new model was not more accurate than the best model among the voters, and the amount of right predictions was at halfway between the performance of each of the models which voted, near to mean value. The variance and bias of voting also depend on the voter models, therefore the variance and bias were near the mean of the 3 models.

But the notion of voting could be really useful when used differently, like with a notion of randomness, and maybe with decision trees ?

III - Modeling: Random Forest Algorithm (Cf. appendix 3)

The model that we finally used was the Random Forest Decision Tree, since it's the model that gave us the best precision. When we know how the Decision Tree model works, this model is easy to understand.

The first step is to create a random subset of our dataset, by selecting a number of lines equal to the number of data in our train dataset while allowing taking the same line several times. The lines that were not used could be used to determine the Out-Of-Bag Error.
After this, we create a regular tree, but with a little trick when we look for the best split. At this stage, we only consider a certain number of columns selected randomly. Therefore, like in every decision tree, we will create decision nodes till we reach the maximum depth, or till the amount of remaining data is fewer than a certain threshold value that we set.
We now have a random tree that will be part of our random forest. By repeating this operation, we will create an ensemble of random decision trees that will form our random forest. And for each set of features, every tree will vote for a label, and the most common label will be chosen.
Since every tree is based on a different bootstrapped version of the original dataset, it's a bootstrap aggregation. This method is called bagging, and it aims to reduce the variance and prevent overfitting, by lessening the impact of special cases in our model. With this method, we can calculate the OOB Error by determining the accuracy of the prediction of the different labels with the trees that were not trained with them.

Our implementation has 4 modifiable hyperparameters. There is, the number of trees that we aim to create, the maximum depth of each tree, the number of columns that we take into account at each time that we look for the best split, and the minimum number of data necessary to split a branch. By running the code several times, it seemed like the best hyperparameters were, a maximum depth of 9, 9 columns when we split, and a minimum of 6 lines to split a node. It's also good to mention that, when we tried our model with the Gini Index, the results were slightly worse than with Shannon's Entropy, but it can decrease the computation time. The number of trees should be as high as possible, in order to reduce the variance, while being careful with the computation time. In our case, we took 1000 trees, and it already took more than two hours to train our model. The final model gave us a precision of 67,83% without preprocessing, and 67,81% with.

We also figured out that our model is rather pessimistic. Even if the 3 labels have the same proportion, this model predicted a majority of Dead labels, almost 40%, while having an accuracy of 61% on this label. Nevertheless, like every model that we have tested, the random forest struggles when it comes to predicting the Strong Covid label, representing 25% of the predictions, and having an accuracy of 65%. And on the other hand, the Soft Covid label is predicted 37% of the time, while having a really good accuracy of 77%.

Conclusion

To conclude, amongst all the models we implemented we witnessed the highest accuracy with the random forest, depending on its hyperparameters. Although, by looking at the Kaggle submission score, the simple decision tree had an accuracy slightly better than the random forest one. It's possible that the simple decision tree is overfitted to the public leaderboard data, since with the remaining 70%, the random forest had a huge improvement in accuracy (from 66,7% to 68%), while the decision tree only improved a bit (from 66,7% to 66,9%). It's a pretty good example of how much we should be careful with overfitting.

It would be interesting to prune the decision tree in order to reduce the number of nodes and prevent overfitting and also speed up the algorithm. Another great validation method would be the cross-validation which we tried but could not run since our models were too heavy and long. The code can be found on Appendix-2.

The Gradient Boosted Decision Tree could also perform better, being ideal for high-dimensional data, but it's also a lot more complex to understand.

Bibliography

1. “AI and Control of Covid-19 Coronavirus - Artificial Intelligence - Www.Coe.Int.” Artificial Intelligence, https://www.coe.int/en/web/artificial-intelligence/ai-and-control-of-covid-19-coronavirus. Accessed 21 Apr. 2023.

2. ARESD 2023 - CovidChallenge. https://kaggle.com/competitions/aresd-2023-covidchallenge. Accessed 17 Apr. 2023.

3. Seaborn.Heatmap — Seaborn 0.12.2 Documentation. https://seaborn.pydata.org/generated/seaborn.heatmap.html. Accessed 17 Apr. 2023.