Thomas Kismarton 6/4/2022
CIS 472 Final Project

Abstract

Three machine learning models – perceptrons, logistic regression, and support vector machines – are employed to determine whether individuals are at high or low risk of heart disease. The workings of these machines are explained, their hyperparameters tuned & tested. Due to homogeneity within the data, the logistic regression and perceptron models default to always guessing low risk, the support vector machine has more interesting and likely more accurate predictions.

Introduction

Within machine learning, the issue of classifying data is a common problem with a myriad of solutions. Various different algorithms and learning models approach classification in many different ways, usually considering and weighing all aspects of complex data points and attempting to determine how that data should be best grouped or represented. A few of these learning models include perceptrons, logistic regression, and support vector machines. In this report, I attempted to use these three different machine learning models to classify individuals as either at-risk or safe from heart disease. This type of classification is known as binary classification, since every person falls into one of two categories – they're either healthy, or at-risk of potential heart failure.

I plan to train all three of these models on the exact same data set, tuning their hyperparameters as necessary to achieve the best possible performance that I can, in order to determine which of these models best achieves the goal of classifying individuals as either 'high-risk' or 'low-risk' for heart failure.

Background

To better understand the difference between these models and their corresponding learning algorithms, I have provided a breakdown of how each of these learning models operates, with a certain level of abstraction:

Perceptron

The goal of the perceptron model is to correctly classify a data point according to the characteristics of that data point. In binary classification, perceptrons only have to choose between two possible values, which can be represented nicely as positive and negative values.

For each data point (**x, y**) that we feed our model, the perceptron takes a weighted sum of all the elements of **x** and makes a prediction of the data point's classification based on this sum. In binary classification, the perceptron compares the sign of the weighted sum against **y**. If the signs match (both positive/both negative), then the model has correctly classified **x**. This process of taking the weighted sum is known as finding the activation of (**x, y**).

If we're in the process of training our model, then each time we incorrectly classify a data point, we update the weights within our model used for computing activation and adjust a value for bias – a variable which helps us shift the location of our decision boundary without changing its shape. If we're simply using our model in testing, then we track the number of correct vs incorrect classifications for later analysis.

Logistic Regression

The logistic regression model operates in a very similar way to the perceptron, in that it accepts a data point (**x, y**) and computes a weighted sum. Instead of simply multiplying the elements of x by some weights, however, a loss function is computed & summed. In logistic regression, we use a loss function known as negative logistic loss, which tells us the likelihood that a certain data sample belongs to a certain classification. By minimizing this loss function, we ensure that our model is classifying data points the best that it can.

A process known as gradient descent allows us to determine our location on the graph of our loss function relative to the data set, and by computing the slope of that graph at the model's current location, we can intelligently pick the way we update weights within the model such that we approach the minimum of our loss function. With enough iterations, the model eventually converges to a certain set of weights & bias that minimize loss.

SVM

Support vector machines combine ideas from both of the previous models, attempting to find a hyperplane that best separates our data into different classes, while adjusting this hyperplane intelligently to quickly reach an optimum.

In the case of binary classification, the SVM attempts to form a single line that splits the data into two classes as cleanly as possible. The model attempts to place this line in such a way that the data points closest to the line have their distance from the line maximized. This distance to the nearest points is known as the margin, and by maximizing the margin, the model has the best chance of correctly classifying unseen data. For linearly separable data, the SVM can cleanly draw this line such that no points are misclassified. This data is not linearly separable, however, and the machine must decide how many points to misclassify in the name of maximizing the margin. A hyperparameter, C, controls the model's preference for classification over margin maximization.

Finally, SVM models use a tool called a 'Kernel function' to transform non-separable data in lower dimensions into separable data in a higher dimension. This allows the model to form a hyperplane that separates previously non-separable data and find non-linear solutions to classification problems. There are several kinds of Kernel functions, each of which performs differently in terms of classification accuracy. The Kernels explored in this report are the Linear, RBF, and Polynomial Kernel functions.

Methods

For each of the three models, I tuned the hyperparameters to achieve maximum accuracy on the validation data before running tests to get the model's overall accuracy on the testing data. The following subsections describe the specific hyperparameters tuned for each model.

Perceptron

The only hyperparameter within the perceptron model is the number of times the model processes the training data, known as the number of epochs. This makes the model relatively easy to optimize, as only a single parameter must be tuned.

Logistic Regression

Logistic regression is less simple. There are three hyperparameters to tune: learning rate, regularization weight, and the maximum iterations.

Since logistic regression eventually converges to an optimal solution, ideally the maximum number of iterations would be set to infinity. This, however, can lead to overfitting, and also takes far longer than is feasible to run. As such, we include it as a hyperparameter for testing's sake.

The learning rate controls how much our model advances down the direction of the gradient in each iteration. Large learning rates tend to result in large swings of accuracy but will eventually converge to the same result as smaller learning rates. Smaller learning rates result in a more controlled approach but may take more iterations to make meaningful progress towards the optimal solution.

The regularization weight controls how much we penalize our model forming high weights for certain variables. This hyperparameter essentially allows us to prevent the model from depending on the value of only a single variable, as increasing the weight of that variable gets increasingly difficult as it attains higher and higher magnitudes.

SVM

Support vector machines similarly have three hyperparameters for tuning: C, the Kernel function, and Maximum Iterations.

Again, maximum iterations are included primarily as a testing parameter, as running the model without this parameter took approximately 2.5 hours. I found, during testing, that the model seemed satisfied with 1000 iterations, after which there were diminishing impacts on the accuracy of the model.

The Kernel function determines how we attempt to convert inseparable data into separable data in a potentially higher dimension. The Kernel functions considered in this project were the RBF (Radial Basis Function), Polynomial, and Linear Kernel functions. To find the optimal Kernel, I simply optimized the other hyperparameters first, before testing each Kernel to see which one produced the highest accuracy.

The hyperparameter C, as explained in the section above, controls the model's preference for correctly classifying data vs. maximizing the margin. Large values of C mean that our model forms a very strict decision boundary that heavily penalizes misclassified data, while smaller values of C mean that the model focuses more on maximizing the margin and generalizing better to new data. When tuning this parameter, I initially did a large sweep, testing all values between [1: 0.1] in increments of 0.1. From there, I took the optimal value for C and did another sweep, this time in increments of 0.01, on the 10 values surrounding the optimal value (5 above & 5 below).

Experiments

The tables of data collected throughout the optimization phase for each model is displayed below. The highest validation and testing accuracy for each machine is highlighted in pale green.

Thomas Kismarton
6/4/2022
CIS 472 Final Project

Perceptron

	A	B	C	D
1	**Epochs**		**Validation Accuracy**	**Testing Accuracy**
2	100		0.639	
3	200		0.818	
4	400		0.584	
5	450		0.786	
6				
7	470		0.836	
8	471		0.843	
9	472		0.804	
10	473		0.756	
11	474		0.846	
12	475		0.913	0.911
13	476		0.733	
14	477		0.806	
15	478		0.836	
16	479		0.886	
17	480		0.779	
18				
19	500		0.893	
20	525		0.899	
21	550		0.848	
22	600		0.826	
23	700		0.7534	
24	950		0.768	

With only a single parameter to tune, I chose relatively random values, using a hill-climbing algorithm to fine-tune the accuracy of the model once it had gotten close. With enough epochs, the perceptron model was able to tune in on a very high accuracy both on the validation and training data. Note that the accuracy of the model declines as the number of epochs exceeds 475. This is due to the model beginning to overfit the data, and the weights inside the perceptron model getting too large. The model begins to depend too heavily on a single variable and fails to generalize to new variables.

Thomas Kismarton 6/4/2022
CIS 472 Final Project

Logistic Regression

	A	B	C	D	E
1	**Max_Iters**	**Learning Rate**	**Regularizer**	**Validation Accuracy**	
2	10	1	0.1	0.915	
3	10	0.1	0.1	0.915	
4	10	0.05	0.1	0.915	
5	10	0.005	0.1	0.915	
6	10	0.1	0.1	0.9153	
7	10	0.1	0.05	0.9153	
8	10	0.1	2000	0.085	
9	10	0.1	200	0.9153	
10	10	0.1	1000	0.085	
11	10	0.1	500	0.085	
12	10	0.1	250	0.9153	
13	10	0.1	300	0.085	
14	10	0.1	275	0.123	
15	10	0.05	275	0.9153	
16	10	0.08	275	0.9153	
17	10	0.09	275	0.9153	
18	10	0.095	275	0.9153	
19	10	0.097	275	0.9153	
20	10	0.098	275	0.45	
21	10	0.099	275	0.199	
22	10	0.0985	275	0.257	
23	10	0.0975	275	0.882	
24	10	0.0972	275	0.9153	
25	10	0.0973	275	0.915	Testing Accuracy
26	10	0.09725	275	0.9152	0.9133

This model was extremely sensitive to change, swinging wildly between 0.9153 and 0.085 in terms of validation accuracy with even slight changes to the learning rate and regularizer variables. I first attempted to at least find the boundary where the value shifts, which occurred around a regularizer value of 275. From there, I wanted to see if there existed a learning rate that exceeded a validation accuracy of 0.9153, and through very fine incremental tuning, I was able to get extremely close to the 0.9153 validation accuracy.

I figured that, at this point, the model had been trained as best as it could, while trying to avoid overfitting. The testing accuracy for this set of variables appeared to be high, but that could have been the model simply predicting that all of these examples are classified as negative.

Thomas Kismarton
6/4/2022
CIS 472 Final Project

SVM

	A	B	C	D
1	**C**	**Kernel**	**max_iter**	**Validation Accuracy**
2	1	RBF	1000	0.773
3	0.9	RBF	1000	0.566
4	0.8	RBF	1000	0.783
5	0.7	RBF	1000	0.901
6	0.6	RBF	1000	0.9
7	0.5	RBF	1000	0.565
8	0.4	RBF	1000	0.569
9	0.3	RBF	1000	0.587
10	0.2	RBF	1000	0.515
11	0.1	RBF	1000	0.502
12				
13	0.65	RBF	1000	0.879
14	0.66	RBF	1000	0.674
15	0.67	RBF	1000	0.879
16	0.68	RBF	1000	0.674
17	0.69	RBF	1000	0.901
18	0.7	RBF	1000	0.901
19	0.71	RBF	1000	0.812
20	0.72	RBF	1000	0.812
21	0.73	RBF	1000	0.875
22	0.74	RBF	1000	0.647
23	0.75	RBF	1000	0.724

	A	B	C	D	E
1	**C**	**Kernel**	**max_iter**	**Validation Accuracy**	
25	0.695	RBF	1000	0.901	
26	0.696	RBF	1000	0.777	
27	0.697	RBF	1000	0.724	
28	0.698	RBF	1000	0.798	
29	0.699	RBF	1000	0.777	Testing Accuracy
30	0.7	RBF	1000	0.901	0.9
31	0.701	RBF	1000	0.811	
32	0.702	RBF	1000	0.747	
33	0.703	RBF	1000	0.811	
34	0.704	RBF	1000	0.782	
35	0.705	RBF	1000	0.907	
36					
37	0.7	Linear	1000	0.199	
38	0.7	Polynomial	1000	0.103	

Since all optimizations were performed using the RBF Kernel function, when switching to the other kernels, we see that performance drops dramatically. Furthermore, we see that the testing accuracy is near that of the other models but increases and decreases far more gradually.

Results

Interestingly enough, the perceptron and logistic regression learning models had roughly the same accuracy. Due to homogeneity within the data, this is likely a result of 90% of the data corresponding to the same output, so making negative predictions for every data sample correctly classifies 90% of it. This being said, the optimizations for logistic regression were made more or less manually in an attempt to counteract this. The support vector machine performed quite differently from the other two models, in that its performance varied more gradually when tuning its parameters. Furthermore, it approached, but did not exceed the accuracy of the other two models.

I'd have hoped to use less linearly separable data for this project to show that SVM performs much better on complex data than Perceptrons and Logistic Regression – that was an error on my part. I do believe that the SVM model would be able to generalize much better than the others, given a greater number of positive examples.

Conclusion

Having made these models and trained them on the data, there are a few things I would change for next time. If I were to attempt a similar project again, I would first analyze the dataset more carefully before building and training models. This would help to ensure that the perceptron and logistic regression models don't end up in the rut that they fell into for this project – wherein they simply classify all the examples in the same way and claim to have high accuracy.

That being said, I enjoyed working with the SVM and scikit-learn libraries. It was interesting to see how the tuning of hyperparameters seemed far more controlled in the SVM model compared to the others. I'd also like to run these projects in Google CoLab or a Jupyter Notebook environment, since I'm more comfortable using visualization software like matplotlib in a notebook format. More extensive projects and tougher problems seem inspiring, and I'd like to continue attempting Kaggle competitions in my downtime.

References

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

“Scikit Learn - Support Vector Machines.” Tutorials Point, https://www.tutorialspoint.com/scikit_learn/scikit_learn_support_vector_machines.html.