**Surgical dataset**

**The Dataset**

The chosen dataset is very robust. It comes with over 10,000 rows, 24 attributes, and a target. The target in the dataset is whether or not the patient experienced a complication after surgery. While working with this data I learned a lot, both about the dataset itself, and how to use the tools that made the project possible.

**Running the Program**

To run the program, you need the IJava kernel for Jupyter Notebook. The entirety of the code is within the Jupyter Notebook and is all written in Java. Included in the notebook are markdown cells which should give an idea of what is happening in the surrounding cells.

**Graphing Library**

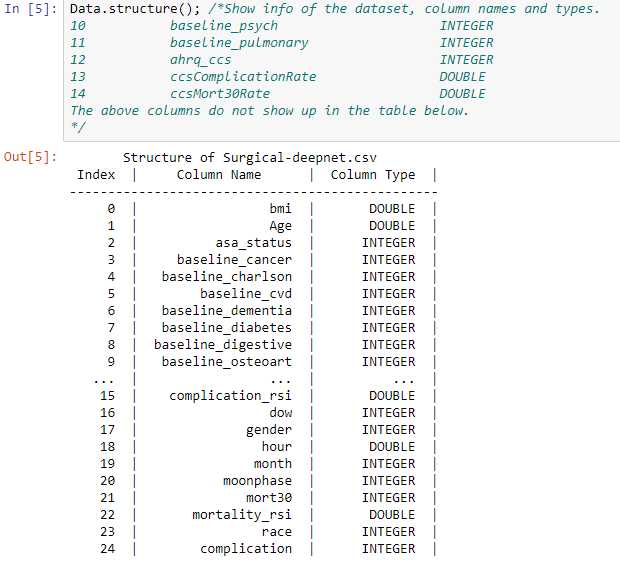
To view the data, I used the Tablesaw library. You should not need to do anything except open the notebook, the first two code cells in the notebook import the library and set the output of the library to be in the notebook itself, instead of creating .html files which it does by default.

**Observing the Data**

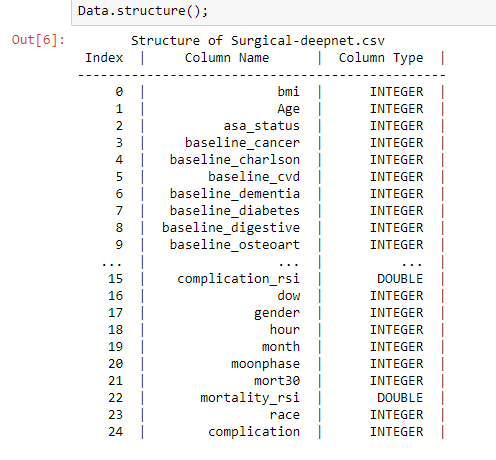
I started out by getting a look at the dataset. Looking at the shape, we can see that we are given 14635 rows, and 25 columns.



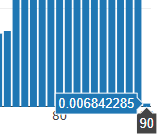
We can also take a closer look at the columns in the next cell, seeing the names of each column and the data type which is stored inside. Here, I also commented the missing lines in the code, as there were too many to be outputted.



After that, I wanted to remove some double columns, by converting them to integers. To do this, I took a copy of each column using the .asIntColumn() function, and then replaced the original columns with the newly generated columns. The columns I changed to integers were only ones with 1 unit of precision, and numbers I thought would still make sense as an integer. The attributes I changed to integers were bmi, age, and hour. The resulting structure is shown below.

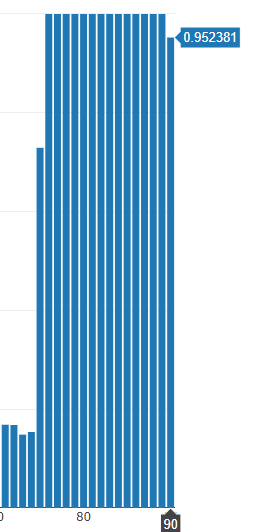


Now that we have cleaned up some of the columns, we need to clean up rows as well. While looking at the data in graphs, I noticed an anomaly. It was most apparent in the ‘age’ column but had big effects in all the data. There were duplicated rows. This resulted in people age 90 having a seemingly incredibly low complication rate.



Noticing this, I manually looked through the data and recognized the issue: duplicate rows. My next step was removing duplicate rows in the dataset. I was comfortable doing this because there are so many attributes, and many of the attributes have large ranges, so the chances that we delete a legitimate case is extremely low. After we deleted duplicates, we get a new row count and complication rate for people age 90:



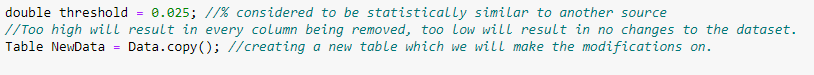


Next, I wanted to graph more of the data, and allow anybody else to graph more data easily. There are two cells in the notebook which have strings “target” and “condition1” or “condition2”. Condition1 can be changed to any integer column, and condition2 can be changed to any column at all. The first graph shows the % of cases where the target is 1, and the second shows the mean of the condition for the given target. This lets us easily see the data in a more relevant state.

After that I created two arrays, one which held the name of each column, and another which held the type of each column. This allows us to easily iterate through the data and make changes.

**Modifying the Data**

Finally, we’re starting to modify the data. I created a copy of the original data table so that the previous cells can be executed again to display new graphs and modified that set. We start off by taking a double value, this double value is used to signify how close two integer columns % target hits must be to be considered similar. This is a completely changeable value and will result in every cell lower getting new data.



We only are considering integer columns here; double columns do not get modified at all. The data is modified in the following way: we find the percent times the target is hit for each value, storing both the value and the double percentage in separate temporary arrays. We sort the arrays based on the percentage time the value hit the target. Our sort is slow, but it gets the job done.

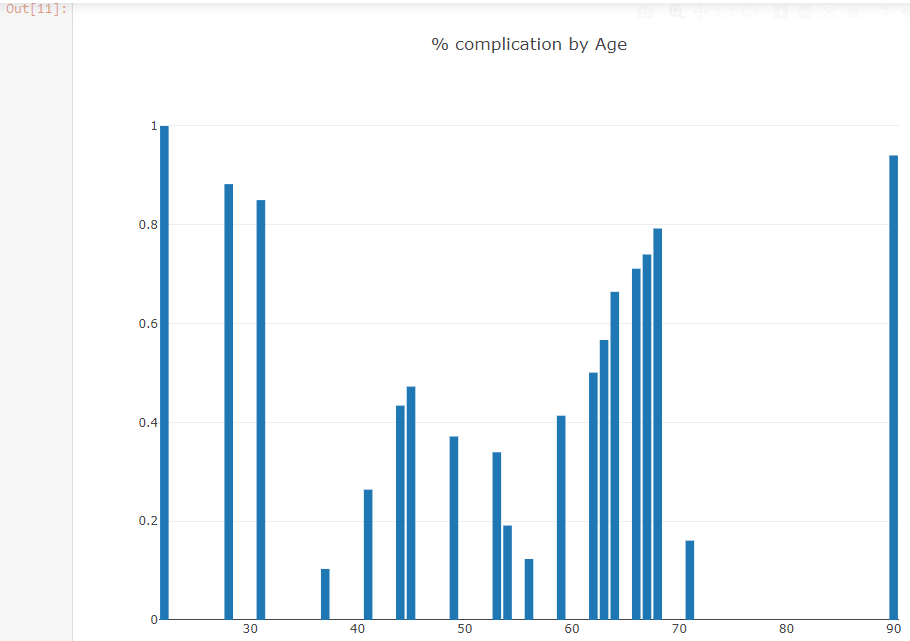
After we have the arrays sorted, we compare the percentage values to each other using the given threshold. We subtract the threshold from the original percentage. The reason I decided to subtract the threshold instead of multiplying the percentage by our threshold (or 1 – threshold) is so that smaller percentage values are not impacted at a different rate. For example, .3 \* .975 = .2925, and 1 \* .975 = .975, I wanted to keep the range in these cases similar.



After we have this temporary value, we compare it down the line. Since the array is sorted, we only need to compare until we find a value which is smaller than the temporary value, as we know every other value in the future will also be smaller. When we find a value which is similar enough to another value, we 3 thing: 1) we change every value in the table NewData from the replaced value to the new value; 2) we set the array value for the replaced value to the new value, and 3) we set the array percentage for the replaced percentage to the new percentage. This means that our ending arrays will have duplicate values.

The reason for replacing the numbers in the array is so that we do not compare new percentages to a new percentage. Consider the case where we have 3 values, each with their own percentage, 100%, 98%, and 96%. What we want is for the 100% and the 98% to combine into one term, and the 96% to stay as itself. If we did not change the new percentage, we would instead have all 3 values become one, which would violate our predetermined threshold of 2.5%.

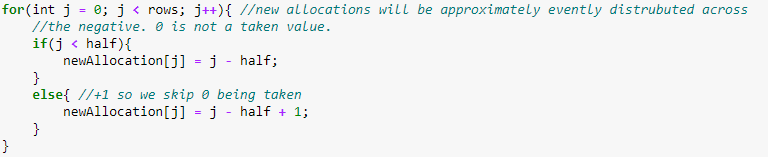
After all of those calculations, we take another look at the age graph, and see that it looks much sparser:



This is the result we were hoping for. Now, we make one more round of changes to this data. We create a new table, Modified, and set it equal to our NewData table, and then change make our final adjustments to the data. In this loop we are iterating downward, since there are some columns which may be removed from the table. First things first, if the column only has 1 value, it’s useless to us, and gets removed from the table.

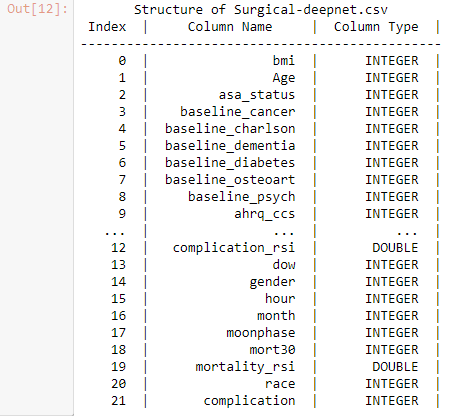
Next, we need change any attributes which only have 2 values to be +1, -1, instead of whatever they were before. We make a case that counts how many values are in the column, and check that neither value is already -1. If neither are -1, we pick a value randomly (I believe it’s the first value to appear in the data) and set it to -1 and set the other value to +1. If a value is already -1, we set the other value to +1.

Finally, we need to do something to the columns with more 2 values. First, we sort the data by percentage, again. With our sorted arrays we go through a loop to see where these values should be put. We want half of the values to be -, and half to be +. So we create a loop which gives each value their new position in a tertiary array.



Now we know what we want the values to become, but first we need to check that there is no crossover. If we turn all of one value into another value which already exists, we’ve mixed the data and cannot recover it. We check that the new position for a value is not yet claimed, if it’s free, we’re happy and make no change. If the position is already taken, however, we instead make the conflicting value turn into the largest number in the column + 1. This ensures that we do not mix the value, which is moving out of the way, but it may be an issue for extremely large datasets. After that , we are free to make the original movement of values, which we do all at once in a loop.

At this point, we’re done manipulating our data, and the structure of the new data can be seen. We removed 3 columns from the original dataset which were not giving us useful information regarding the target, the final structure can be seen below.

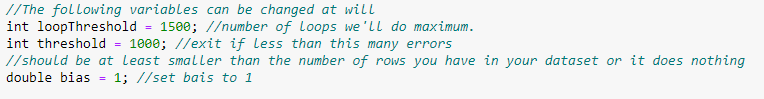


**Creating the Weights**

We start by splitting the data into two sets, a training set and a testing set. The training set contains 80% of our data, and the testing set contains the remaining 20% of the data. We can see the size of the training set after creating the table.

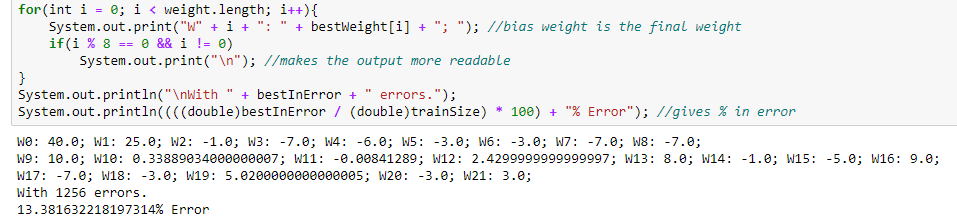


For training the data there are 3 changeable variables, the loopThreshold, which is how many times the algorithm will loop maximum, the threshold, which is how many errors we will accept to stop the loop early, and the bias. Any of these can be changed to other integer values. The algorithm usually has pretty good weights after 500 iterations, but the default value is 1500 iterations (please excuse the spelling error in the notebook).



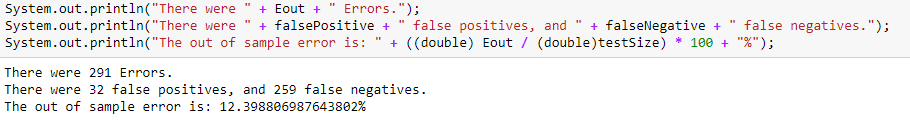
The weights are all initialized at 0, and then implement the pocket algorithm. For each row we check if our weights \* attribute gives us the sign of the target, if it does, we continue, if it doesn’t, we increment our error, and add the row to our error array. After we check every row with the given weights we continue, replacing the weights in the bestWeights array if our error is better than the best we have recorded (initialized at a high number so the first iteration always copies over). Next, we break if either of our conditions have been met, either too many iterations, or an acceptable in sample error.

If we are still running the loop, we create a random number of maximum size equal to the number of rows in the training set. If this number has an error (according to the error array), we keep this number, otherwise we repeat until we find a row where an error exists. We use this row to update our weights. I also found that setting the new weights = bestWeights \* attributes gave better in sample errors than just setting the weights = weights \* attributes. We continue the loop until we exit due to one of our previously set conditions. We then output relevant information, displaying the # of errors, and percentage of errors.

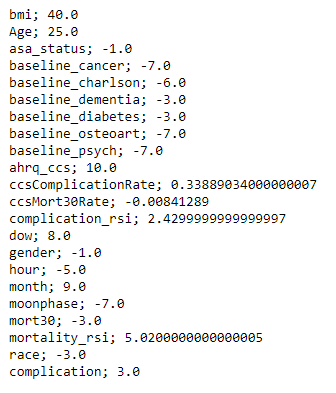


**Testing our Weights**

Now that we have our weights, it’s time to test them. To do this, sum the weights and attributes, then find if the sign matches the target. In addition, I also kept track of the types of errors we receive. I’m pretty happy with the out of sample error, and think our weights work pretty well.



After checking the out of sample error, I did one last thing. I outputted the attribute names, and their weights. We can see which attributes are the most representative of the target by looking at the magnitude of the weights. The patients BMI and Age seem to be the most important parts when determining their complication after a surgery.



**Difficulties**

I encountered a lot of difficulties completing this project and learned a lot in the process. The biggest difficulties I encountered were regarding using Tablesaw. This is the first time I’ve ever used both IJava and Tablesaw, so learning how they interacted was a struggle at first, but luckily, I found a good post on stack overflow which helped me implement Tablesaw in IJava. Tablesaw also thankfully has very robust documentation. I was able to find everything I needed to manipulate the tables and columns as I needed, though some things took a lot more digging than others.

Overall, I learned a lot in the process, most of my errors were simple mistakes that I kicked myself over, but I’m more cognizant of those mistakes now, and won’t make them as much as before. The biggest lesson I learned from this was how to use other people to find solutions to your mistakes. Even when I didn’t know what the compilation error meant, looking for other people with similar issues allowed me to locate the problem in my code, and find an amicable solution.

**Closing Remarks**

I tried to make this program work quickly for any dataset with only Integer or Double values, and I think I was successful. The Jupyter notebook file should be very straightforward and explain everything that’s happening as it happens in the code. This project was a lot of fun, and I look forward to doing more machine learning in the future.

**Conclusion**

We were able to create a pretty good set of weights for the given dataset. We started off looking at how the data was formatted and converted a few double columns into int columns. We proceeded to see graphs of the data, allowing us to easily look at any given attribute in relation to the target. Finally, we combined data which hit the target function a similar amount of times, and completely removed columns which only had 1 value after the combination. Finally, we gave the columns completely new data, ordering it in terms of probability to have the target function be a 1, and then put half in the negative class and half positive. After completing data manipulation, we ran the data against pocket algorithm to find weights and calculated our final Ein and Eout. We can also see which attributes impact the prediction the most by observing the magnitude of the weights.