Utilizing Random Forests to Identify Features in Patient Clinical Notes

James Cavallaro

Michael Wong

Xin Shen

Abstract:

Our problem was to find a way to automate the grading of a note taking assignment for doctors. The student doctors would be given a mock patient who would describe an issue they were having, while the doctor took notes; these notes would then be analyzed and graded based on which features from the script were mentioned in them. Our goal was specifically to find a way to highlight these features in the notes, getting essentially one step shy of fully automating the grading. Our approach was to use a random forest, utilizing the given training data to teach it when and where to mark the start/end of a feature.

Introduction:

As part of the Step 2 Clinical Skills examination, a subsection of the greater United States Medical Licensing Examination, test-takers are required to be involved in a mock appointment with a Standardized Patient (a test administrator posing as a patient). In this appointment, test-takers must write patient notes, describing the patient's condition as well as their symptoms. After the examination, these patient notes are graded by trained physicians from within the field. The notes are evaluated based upon how well they describe the concepts, or features, of their given patient. In general, the more features of the given case that the note processes, the greater the score. However, the issue with this form of examination is the inefficiency introduced when grading. Scoring these notes consumes far too much time and resources. Thus, the problem that our project is attempting to solve is the optimization of grading. Using machine learning, we will train our model upon a given test set, matching various descriptions to given features.

Related Work:

There are quite a few articles among Medium, TowardsDataScience, as well as other websites that describe similar implementations of Random Forests in python. Some manually build Trees and aggregate their forests, others use similar libraries used in our project, such as pandas and sklearn. Structurally our approach is similar to some of these articles, whether they use the same libraries or not. We first create a data set, prep it and divide it into training and target sets. Then we build our model and then train it. Finally we adjust. Where our approach drastically differs is how the data is prepped. Because of the string nature of our data and objectives, much more care has to be put into preparing the data set. Since machine learning models generally do not use strings, our string data had to be separated into lists, placed into individual columns, and ran through a one-hot method (this will be further discussed in later sections).

Data:

The data provided to us by the Kaggle competition was split into two csv files: one for the patient notes, and one for the features. The latter consisted of the features present in each of the 10 cases; the former consisted of over 40,000 patient note histories across each case, and was the data to be graded. There was also a train file, consisting of 1000 patient note/feature pairs, 100 for each case: more specifically, each entry contained the ids for the case, feature, and note history, the text of the noted feature (if it was present), and where in the history that feature was mentioned (if at all). For example, if one of the cases had "hair loss" as a feature, if the patient note mentioned hair loss, the entry in the train file would contain the exact wording in the patient notes and the location of the first and last character of it. If it was mentioned multiple times in the notes, it would appear multiple times in the entry- the columns for feature and location were both arrays. If it did not appear at all in the notes, the arrays would both be empty. There was also a test file mentioned in the competition, containing 2000 unannotated histories from the patient notes file, but the actual train file seemed to merely be a small sample input

Our approach was fairly simple; use the train file to teach the algorithm how to recognize the features in the history, and how to note the character span for each. Then, test it against the data to see how to improve. While a test file would have been nice, it ultimately would have been just a time saver. There was no special treatment needed for the data before being put into the algorithm.

Methods:

Our primary tool for this project was random forests. The main reason for this is that a random forest would allow for categorization. With a random forest, we can easily have the algorithm identify a particular phrase with a particular feature, allowing it to know if something is a match or not. From there, we can simply get the location for the first and last character of the phrase to get its character span. The primary reasons to use a random forest over other decision tree methods was noise and the training process itself. The patient note histories are inherently noisy, as they contain a lot of "fluff" that is irrelevant to our goal, but cannot simply be discarded; a random forest is better at handling this noise than other tree models. Additionally, with the way the data had been given to us, bagging was a natural way of training the algorithm, as it was very easy to simply take a random entry in the train file for each step in the training process.

Theoretically, a k-means clustering algorithm could have also worked for this project; however, there are two main problems with this. The first is the aforementioned noise- not only is not every word within the histories relevant, but the algorithm also has to recognize when a group of words belongs to a single feature, and variants on the phrasing of those words. The second is the data itself, and specifically how it is structured. Our goal was not to organize data into one of multiple categories, but to recognize if a data point exists *at all* within a given block of text. A clustering algorithm would not be a good fit at all for the data we had unless we modified it massively, and with the amount of data to go through, that would have been a massive time sink to do.

The first challenge of this project was prepping the data for our model. As per the kaggle competition, we were given a .csv file containing data for a training set. This data was opened into google colab with the data management library Pandas. With our data properly organized, it was ready to be modified for the random forest. Since the model that we are using is made with the Sklearn library, which will be discussed further in the next paragraph, our rows and columns, barring their names/titles, must only contain numerical data. Thus this creates a problem, as our competition is meant for matching string data to string data. However, this was not the only issue found with the data. The ‘annotation’ column of the training data, of which we are supposed to map words to a note feature (the distinction should be made that a note feature is the category of a note in the USMLE examination and not a ‘feature’ as is known in machine learning), contains a whole string phrase that is related with a corresponding note feature. This would become a problem for matching the phrases, because if the phrase doesn't match the string exactly, the model may not be able to correctly determine its relation. Therefore the data must also separate the phrase string into individual word strings. In order to tackle both of these problems, we combined string splitting with a One-Hot encoder method. String splitting is fairly self explanatory. We split the strings in the 'annotation' column by white spaces and insert them into a new column called 'dictionary'. This 'dictionary' column contains a list of words, each with an individual string. From there, we use the MultiLabelBinarizer from the sklearn preprocessing library to One-Hot encode the data. One-Hot encoding ensures that we have only numerical values while still being able to reference our string data. This Binarizer takes our column full of lists, expands/appends each word as columns, and One-Hot encodes them. Our final step of data prep was dropping the redundant data columns in 'annotations' and 'dictionary'. With this, our data is fully ready for the model.

The second challenge was creating the random forest. Utilizing the Sklearn library, we were able to more efficiently build out the model. Sklearn has an effective, prebuilt classifier called the RandomForestClassifier. Using this in conjunction with the data previously prepped, we were able to divide the training data into two sets: training and test. Our test data contained 25 percent of the training data. The 'y' of the data is the 'feature_num' column, with the 'X' being the rest. From there the RandomForestClassifier was used to create the model. With n_estimators, the number of trees in the forest, being only 5, the accuracy of the model was at 18 percent.

Experiments:

As mentioned in the previous section, the accuracy of the model was extremely low. This is due to google colab's resource restrictions on free accounts. Without enough memory, the model was not able to complete its training. Moving over to Jupyter allowed for the model to be ran on local hardware. Thus we were able to increase the amount of trees in the forest. With 20 trees we began to see progress. The accuracy hovered around 50 percent. We continued to increase the trees to moderate and drastic lengths. With one hundred trees, the accuracy grew to roughly 60 to 62 percent. At two hundred trees it increased to around 63 percent. At one thousand trees the accuracy was at a solid 64 percent. The trade off between the amount of trees and accuracy was minimal at the more drastic ends. Therefore we settled on our model containing one hundred trees. We then experimented with individual max tree depth. At low numbers, such as 10, accuracy yielded at a mere 50 percent. In the case of tree depth, extreme numbers favored accuracy while not detracting too much from run time. Thus we settled on 1000 max tree depth. So with a tree count of one hundred and a max tree depth of one thousand, the accuracy was stable at 63.9 percent. While not the most ideal accuracy value, with considerations to the leaderboards on the kaggle competition, of which actual data scientists boasted a max score of 89, the accuracy for our first ever Machine Learning model was presentable.

Conclusion:

We used random forests to teach it when and where to mark the start/end of features given the training data. We created a dataset and divided it into a training set and a target set. Then we build our model and then train it. When we increase the number to 20 trees, we can see considerable progress, with an accuracy rate of about 50%.When we have a hundred trees, the accuracy grows to about 60% to 62%. And on 200 trees, it increased to around 63%. On a thousand trees, the accuracy rate stabilized at 64%. The conclusions we get at this level are impressive, meaning our code works and our ideas are correct. We make it easy for medical students to display important words such as "heart, pain, duration" with the greatest convenience when recording. When taking the test, the time for manual review of notes can be reduced to a greater extent, and the test can be completed more efficiently.