Fall 2021 **Final Project Report Total = 65pts**

**Introduction**

This project ended differently than it began. I began with the idea of translating error messages to Spanish, then shifted to classifying python compiler error messages, and ended up classifying C compiler errors (see *Analysis* section for why). The problem of classifying compiler errors was a small step towards making compiler errors more useful. Compiler errors are notoriously cryptic and not very useful (especially for beginner programmers), so almost any improvement would be helpful. By classifying the errors, the goal was that the solutions for errors within each class of errors would be similar to the solution for other errors in that class. That way, if a user knows how to fix even a single error within a class, they could have a rough idea of how to fix any new error that arises that has the same class.

The most useful paper I read (and re-read) is the one I ended up getting my dataset from. This paper (Ahmed et. al, 2019) was about how to take new programmers’ compilation errors and automatically correct them. The idea was that a SVC would record every single-line error that a student made, and subsequently the corrected line once the student uploaded a working version of the code. This allowed the authors to categorize different types of errors based on how they were fixed. The authors used a dense neural network (DNN) to organize error classes. The authors of this paper did not actually generate the dataset, and so I have also cited that research team (Das et. al, 2016).

**Model**

My goal was to create a deep neural network to classify the errors, because sometimes the error belongs to multiple error classes, making this a multi-label classification problem. Most of my reading suggested that for multi-label classification, only NNs gave consistently good results. However, I was unable to complete a working model, so I cannot say how accurate the model was (see *Analysis* section).

As an example, consider a neural network that is meant to identify which colors (out of a given list, say [green, blue, yellow, black, red, purple, orange, white]) are present in an image. First, the NN would need to be trained. This is the step during which the NN analyzes the features of a bunch of labeled images, to determine which features are the most useful for identifying colors present. The data needs to be already labeled so that the NN can associate and weight different features to labels it knows are correct, and it continually updates these weights as it is trained on the data. Eventually (ideally), the NN has weighted all relevant features to the point that it can be given new images, for which it doesn’t already know the labels, and predict the colors present.

**Analysis**

The biggest reason I switched from classifying python compiler messages to C compiler messages was due to data availability. I simply couldn’t find any dataset robust enough to analyze of python messages. I was able to find a dataset of C compiler error messages, from [[[[]]]] mentioned in the *Introduction* section. I thought this dataset would work well for several reasons: (1) it was large, with over 20,000 data points; (2) the faulty programs had been compiler by two different compilers, Clang and LLVM, which each gave different error messages than the other; and (3) the dataset included “error classes” that were associated with each compiler error message, that the authors of the paper had constructed of their research. The large size of the dataset would ensure that even after cleaning the data I would have enough datapoints to train my model; in fact, the useable data was just over 7,500 data points. Having error messages from two different compilers would give the model more data to work with and ideally more features, or at least features that were more relevant to classification. Lastly, because the authors had associated each datapoint with one or more error classes, I would be able to train my model. I almost went with a different dataset, but it had no labels, and so I would’ve been doing more of a clustering analysis than classifying analysis, because I would need to use unsupervised learning.

Ultimately, I wasn’t able to configure the DNN to produce any relevant output classifications. I feel that I understood NNs well enough, but implementing it in code proved more challenging. Therefore, I don’t have any performance measures do specifically discuss. My reading suggested that accuracy isn’t a very good measure for this type of problem, and a better indicator would be the AOC, so that’s what I had planned to do.

**Conclusion**

Overall, I wasn’t very successful at the project, because I couldn’t create a working classifier. However, I learned a lot more about many of the topics we covered in class, especially NNs. I would’ve been interested to see which features proved most relevant for classifying the messages.

**Reflection**

One of the aspects I most enjoyed about this class was the mixture of high-level and low-level thinking. The broad topics that we covered in class were mostly high-level, in order to understand the concepts, while the labs and the project required us to think at the level of the code. Given 6 more months, I certainly could’ve completed and fine-tuned my model, and I would also have pursued how each error class could be addressed (for the coder who encounters the error, how they can more clearly understand the cause of the error and efficiently go about fixing it).

Sources

R. Das, U. Z. Ahmed, A. Karkare, and S. Gulwani. *Prutor: A system for tutoring CS1 and collecting student programs for analysis*, arXiv, 2016.

U. Z. Ahmed, R. Sindhgatta, N. Srivastava, and A. Karkare. *Targeted Example Generation for Compilation Errors*, “The 34th IEEE/ACM International Conference on Automated Software Engineering”, 2019.