Sentiment Analysis in Julia Project Report

Andrew Lee
andrewlee733@u.boisestate.edu

Josh Coward
joshcoward@u.boisestate.edu

November 2020

Abstract

Julia is a high level, high performance, general purpose programming language designed with computational science in mind. We chose this language because it is often used for machine learning and data science and we wanted to compare it to Python through a machine learning project. The project goal was to build a sentiment analysis model for Twitter posts using libraries available in Julia. This report is divided into two sections: the first detailing the project and the second providing a general commentary on the Julia language. Though we did not hit all of our targets, we believe we successfully learned and applied the Julia language to create an interesting project.

1 Introduction

Machine learning and data science are at the forefront of cutting edge technology now a days. They have huge potential when it comes to automating tasks in order to help solve problems around the world. Areas like computer vision, natural language processing and reinforcement learning are a few of the most popular and for awhile they have been mainly programmed in python and a few other languages like R and Matlab. Trying to step outside the norm and use a newer general purpose language designed for computational programming we chose Julia for the project.

2 Project Description

The goal of the project was to show off Julia's computation abilities and speed by creating a machine learning project that used a large data set so that we could see if we could still have high performance on a large data set. After some research we were able to find a large data set that could be used to build a machine learning project out of, and this was the Sentiment140 data set. The data set contains 1.6 million tweets labeled 0 for negative and 4 for a tweet that contained a positive sentiment. The data set seemed perfect for building a sentiment analysis model so that's what we planned on doing with it, with

the end goal of incorporating it inside a GUI for a user to be able to use so they could explore the sentiment of their sentences and phrases and find similar phrases to theirs using the word2vec model created by our tweets from the data set. The idea was to develop multiple models to compare their results with one another to demonstrate Julia's supposed fast development. The initial goal was to also explore the multi-threaded capabilities in order to speed up run time by processing data at a faster rate and also explore the plotting libraries to gain more information from our data but unfortunately the time did not allow for those to be explored and implemented fully.

3 Program Description

Our program consists of several parts, and with Julia being a newer language that is supported by Juypter notebooks, the program is almost entirely written in Juypter notebooks and has the file extension .ipynb rather the .jl for the typical Julia file, The reason for this is Juypter notebooks creates a nice and clean environment at makes it easy to explore and visualize data while also having the ability to run select code block for an easier debugging experience. The project/program can be broken into five section, Data Exploration, Word2vec model creation, the Support Vector and then the Naive Bayes machine learning model creation, and lastly the Graphic User Interface (GUI).

One of the things that made working with the Julia programming language very easy to use coming from a python background is the ability to quickly load libraries in and use their functions for whatever you need them for and this was no exception for the Data exploration portion of this project. Very similar to Python's Pandas library, Julia has its own libraries called CSV and DataFrames, which allow you to explore and manipulate a csv file and its contents very easily. Through the use of these two libraries we were able to explore our found sentiment140 data set and remove all the missing data and extra columns that provide no classification power. This left us with a clean data set consisting of only tweets and their corresponding sentiment values, from their we simply had to clean the individual tweets themselves in preparation for our word2vec model which consists of removing all non alphabetical characters, any tagged users, and all stop words(any word that doesn't add any meaning to the sentence such as he, was, a, have.) from the tweet. With our now cleaned and "denoised" tweets we are now able to build a Word2Vec model.

Word2Vec is natural language processing technique used for understanding words, their context and meaning along with possible synonyms and phrase completion suggestions given an incomplete sentence or phrase. It is built using a neural network that takes in a large corpus of text (in our case this consisted of all our tweets concatenated together) in order to build a usable model that can produce feature vectors for any given word in the text corpus that can be later passed on to a support vector machine learning model. These feature vectors that it produces consists of a nx1 dimensional vector where n is the number of features and the data values consists of floats which together represent the

meaning of a given word in the context of all other words in the text corpus, but more specifically in context to the words surrounding a given word in the text corpus.

With our Word2vec model created, the next step in the project was to actually build our sentiment analysis model using our tweets and their corresponding feature vectors created by the word2vec model. For choosing a model we did quite a bit of research and narrowed down a list of the best possible models for our given classification task these model consisted of Support Vector Machine (SVM), Logistic Regression, Naive Bayes and an RNN. From our research and our conceptual understanding of the problem and the models, the SVM model seemed to be the best choice by far for this problem. SVM's represent data in high dimensional space and attempts to find an optimal hyper-plane that creates the maximum distance between points in order to cluster data given that they have enough similar properties in common with their surrounding data points. An example of this can be seen through imagining a graph, at the center of the graph imagine their is point labeled mother, and to the right of that point you have a couple of other points with the labels girl, female, women, but to the left of the original point mother you might find the points/words parent, caregiver, or father. In some other part of this graph you might have the point representing the color purple and some of the surrounding points might consist of things like, blue and red, or plums, eggplants or any other purple item. In other words SVM's are essentially grouping/clustering words and their characteristics/ meaning together, and this is done in very high dimensions in order to represent tons of ideas, words, and concepts. This concept resonated with us as it seemed like clustering similar positive and negative words together would produce a really good sentiment analysis model as we could picture the results of the SVM in our heads.

Using our Word2Vec model we created a list of feature vectors from our cleaned Tweets along with there given sentiment value and used them to train a SVM using one of Julia's pre-built SVM libraries called LIBSVM. This is where are troubles really began, we started by training on small subsets of our data set initially only using 10,000 of the 1.6 million tweets in order to prove that it was actually training and building a working model. Even with this very small subset of data it would take about 15 minutes or so sometimes to train only to get a model with an accuracy of around 70%. While 15 minutes may not sound like a huge deal, it became a big deal when training on anything above that small subset. At 100,000 tweets the model took anywhere from 1.5 to 3 hours to train to only get an accuracy of 75%, and when training on the full data set it took a minimum of 7 days to train with 9 days being the longest. Unfortunately for us, when the model finished training in Jupyter Notebook, the kernel had restarted as soon as it finished so the model was no longer stored in memory and all the data was lost. Somehow we had enough time to fully train the model 3 times and regardless of what we changed we were never able to save the actual model. This is probably due to the fact that our hardware is simply not capable of processing such a large model with so much data as SVMs are typically very computationally heavy. While we hoped to get the accuracy

of our model up to 90% accuracy it was more likely to be around 80% to 85% if we successfully trained the entire model, but we cannot know for sure.

With the SVM model we switched to a backup plan which consisted of implementing a Naive Bayes machine learning model from scratch, which would be less complex and way less computationally heavy. This was an interesting process as in Julia there are no "classes" like you see in most object oriented languages but instead you have structs like you do in the C language, which was pretty odd to see being that the language seems to steal a lot of things from the Python programming language. With that said fortunately we both have experience using structs so coming up with a Naive Bayes ML algorithm wasn't overly difficult. With our algorithm built all that was needed to do was pass in just the tweets and their corresponding sentiment values (no need for the feature vectors created by word2vec anymore) and from that we were able train and save a sentiment analysis model with an accuracy of about 80%.

The last thing left to do once our model was created and saved to a file was create a GUI to allow users to identify the sentiment in their own provided tweets, phrases, or sentences along with the ability to find tweets similar to theirs in the sentiment140 data set. This was accomplished using Julia's provide Gtk library. The Gtk library provided the ability to create, edit, and display various widgets(buttons, labels windows, etc.) in a desktop interface. This is how we were able to make a interactable interface with our model. While the Gtk library provided some nice features it was quite limited in comparison to other implementations of it in other languages like Python.

4 The Julia Programming Language

4.1 Overall Experience

The experience using Julia was mostly positive. One of the first things you notice about the language when using it is that it combines syntax and features from many different languages, particularly C, Python, and MATLAB. Defining functions, loops, and if-else statements is very similar to Matlab in that you have to always end everything with the keyword "end". It also has a very "plug and play" type feel similar to Python, where everything is designed to be quick and easy to use. At the same time it is designed in a way where it runs like C, fast and efficient and as mentioned earlier it uses the same struct syntax to create user defined composite types. Combining features from many languages along with its intuitive, simple syntax made Julia pleasant to write in, prototyping was quick and program readability was very good.

4.2 Language Properties

4.2.1 Naming and Binding

Binding in Julia is similar to most dynamically typed languages. Binding happens at runtime and because of this Julia does not expect references to be bound

in a specific order. For example, you can define a function that references a variable that has not been declared in the code yet and Julia will not complain so long as the variable is defined by the time the function is called. Naming in Julia is somewhat interesting, since the language allows Unicode characters to be used in variable names. You could have a variable name typed with Chinese symbols or even one with emojis. We never ended up using this in out project, but we thought it could be useful for programmers with mathematical background to be able to use Greek letters as variable names.

4.2.2 Scope

Julia is statically scoped just like most modern programming languages, meaning variable scope depends on the structure of the program.

4.2.3 Types

Julia's type system is very simple and easy to use. In Julia, every object has a type, which all inherit from the generic "All' type. There are both abstract and concrete types, all concrete types must directly inherit from abstract types. This means the type tree is almost entirely composed of abstract types, with the leaf nodes being concrete implemented types. Since Julia is dynamically typed, you do not need to specify type when declaring variables, however you can specify expected types with the "::" symbol. Julia allows you to create primitive types (i.e. types that are just a sequence of bits), though it is not recommended and most often composite types are used instead. Composite types are immutable by default to improve efficiency, but you can also create mutable types with the "mutable" keyword.

4.2.4 Functions

Julia has many interesting features for functions. The normal syntax for function definitions is similar to many other dynamically typed languages, but Julia also has a shorthand notation for functions. This shorthand allows you to write functions on a single line (e.g. "foo(x,y) = 2x + 1"), and looks similar to notation in mathematical formulas. Functions are objects just like any other variable, so you can pass functions through arguments and return values of other functions. Taking a page from functional programming, Julia allows you to chain functions together with the "" symbol. For example, "(f g)(x)" is equivalent to "f(g(x))". Also similar to many functional languages, Julia discourages creating side effects in functions, if a function modifies any of its arguments it is standard to put a "!" symbol at the end of its name to denote that it causes side effects.

4.2.5 Multiple Dispatch

In Julia, functions are defined as objects that map a tuple of arguments to a return value. Methods are a single possible behavior of a function. Functions can have one to many method implementations. Julia uses multiple dispatch to

dynamically map function calls to specific method implementations at runtime. This is different from function overloading in statically typed languages. Static languages know argument types at compile time so they can map function calls to the correct function during compilation. Since Julia is dynamically typed it must do this in real time during program execution. Multiple dispatch is perhaps Julia's most important feature as it is what allows Julia to have highly abstracted functions, while also having highly optimized method implementations for specific types.

4.3 Downsides

There are really two big downsides with this language, the first is that being that this is a new language the support for it is nearly non existent, being mainly a Python programmer you get used to being able to look up nearly every problem you have on the internet and finding some solution for it very quickly where that is anything but the case in Julia, which as a silver lining it forces you to become a better programmer, however it is nice having the safety net of StackOverflow when you're in a pinch. The other downside while sort of related to the first is the lack of libraries and under documentation. While Julia boasts about all their great ML and data visualization libraries they have, for the most part they really aren't that useful and are no longer compatible with the language especially the data visualization libraries due to developers not keeping up with the languages updates. Unless you want to build every ML algorithm you decide to use from scratch Julia really isn't a good language it seems for the purpose of building ML applications and using something like Python Scikit-learn library would be way more beneficial and easier to implement. The ability to save and load models is also pretty much non-existent with the current outdated implementation provide in Julia which basically makes it a completely useless language when it comes to machine learning since you can't even save or load models unless you write the code to do it yourself. This was ultimately very surprising to see after all the belief was going into this project that it was a great language for ML and even considered to be the future language of ML and while it certainly is built for it when looking at from the perspective of a researcher who wants to build a fast computational heavy ML algorithm from scratch, but from the perspective of someone who wants to use a ready to go ML algorithm in order to create useful applications it just falls short of expectations.