Natural Language Processing, Topic Modeling, and Machine Learning with Apache Spark

DATA 603: Platforms for Big Data Processing

Aaron Tan

Introduction

Natural language processing (NLP) is a field of research at the intersection of linguistics and computer science that investigates the capacities of computational algorithms to understand the mechanisms of human language. In recent years, NLP has become an integral part of data science with regards to extracting meaningful insights from unstructured text data. The NLP analytics pipeline involves many stages of iteratively breaking down text into components such as tokens and lemmas. From these components, algorithms are then applied to produce summaries, generate novel text, analyze sentiment, and model key topics. Due to the abundance of text data and the many stages involving in NLP processing, parallelized cluster computing is often essential to carry out the necessary analytics in an efficient manner. In this paper, I will focus on the implementation of topic modeling – specifically Latent Dirichlet Allocation (LDA) – within the framework of Apache Spark. I will discuss Spark NLP, the Spark library specialized for executing NLP preprocessing stages such as tokenization and lemmatization. I will then discuss LDA topic modeling within the Spark MLlib library. I will explicate the functionality of these libraries using the example of the Yelp Reviews Dataset, a 6gb dataset of unstructured text that demands the computational power of parallelized cluster computer and an API such as Spark in order to successfully process in full.

Distributed Computing and Apache Spark

Before delving into the details of Spark NLP, I will discuss the significance of distributed computing in general, and then Spark’s unique contributions to the field. So – why distributed computing? Around 2005, due to limitations in manufacturing technology, hardware developers stopped making individual processors faster and switched to adding greater and greater amounts of processors to computer chips. Simultaneously, data storage technologies kept on growing, allowing for the storage of vast amounts of data and the emergence of the big data paradigm. This situation meant that data collection became increasingly inexpensive, while data processing encountered a host of challenges that required the development of novel hardware and software solutions.

Before Spark, Hadoop MapReduce was the dominant parallel processing engine for clusters, but it was difficult to use it to build large applications, as each pass over the data required the writing of a separate MapReduce job. Spark, instead, was designed to succinctly express multistep applications, which is what led it to its currently dominant position.

Apache Spark is an open-source unified computing engine plus a set of libraries for parallel processing on computer clusters. First, Spark is unified, in that it utilizes a consistent set of application programming interfaces (APIs). Before Spark, when working with distributed computing, programmers often had to piece together multiple APIs in order to successfully build an analysis pipeline. The unified nature of Spark allows the development process to occur much more efficiently. Second, Spark is a set of libraries that can accomplish tasks such as data loading, working SQL-structured data, and working with machine learning pipelines involving pandas and scikit-learn.

In short, Spark manages and coordinates the execution of these tasks on data across a cluster of computers. This coordination involves two parts, the driver and the executors. The driver is responsible for maintaining information about the Spark Application, responding to a user’s programming, and distributing work across executors. The executors are what actually carry out the work—that is, execute the code and then report the results of the computations back to the driver.

The parallelization of these tasks depends on the notion of a partition. For example, in the case of pandas, the data frame is partitioned into collections of rows, each of which is assigned to one physical machine in the cluster.

Somewhat uniquely, Spark is intentionally limited to being a computing engine—it is not meant for permanent storage. For storage, Spark integrates with distributed file systems such as Azure, Amazon S3, Hadoop, and Cassandra. By contrast, Hadoop is both a storage system (the Hadoop file system) and a computing engine (MapReduce).

Another benefit of Spark is that it can be run within many programming languages, such as Python, Java, Scala, R, or SQL. For this course, we will be using Python in the form of PySpark. Spark can be run locally or via a web application such as Databricks. Web applications allow for the usage of large clusters of servers, but they must be paid for. Databricks offers a free Community Edition service, but it has less computational power than a standard laptop. Therefore, the ideal pipeline might be to build Spark data processing pipelines on one’s local computer, and if there is a need for greater computational power, that code can be easily transferred to a web application and run on server clusters.

Spark’s use of ease, flexibility, and wide scope of applications has made it the most popular tool for conducting distributed computing over big data, and a highly valuable resource for navigating the data science professional world.

Spark NLP

The Spark NLP library was developed by John Snow Labs in order to integrate NLP functionality into the framework of Spark’s distributed computing platform. Spark NLP is an independent NLP library (i.e., it does not rely on other popular libraries such as Natural Language Toolkit (NLTK). It contains functions such as: Tokenizer, Normalizer, Stemmer, Entity Extractor, Date Extractor, Part of Speech Tagger, Entity Recognition, Sentence boundary detection, Sentiment analysis, and Spell checker. Spark NLP is a major optimization over previous NLP strategies, such as attempting to integrate spaCy with Spark. The problem is that “splitting your data processing framework (Spark) from your NLP frameworks means that most of your processing time gets spent serializing and copying strings.” Running a spaCy NLP pipeline within Spark requires going back and forth between Spark’s Tungsten optimized format and spaCy’s Python NLP processes, which eliminates the performance benefits associated with Spark’s caching and execution planner. This ends up requiring at least twice the memory and also does not improve with scaling, negating the major purpose of using Spark and distributed computing. Instead, Spark NLP frictionlessly reuses existing Spark libraries, such as ones from Spark MLlib. This way, the entire NLP processing pipeline occurs within Spark’s platform that is optimized for distributed computing.

Spark MLlib

Spark NLP and MLlib in Action: The Kaggle Yelp Dataset

For my project, I am working with the Kaggle Yelp dataset – in particular, the file that contains the text of the user reviews. This file is 5.89 GB and contains 8,021,122 reviews. The project requires a dataset of at least 500 MB, which would mean around 680k reviews. However, given the complexity of processing natural language data, a dataset of that size would take far too long to process on a personal computer. Thus, I chose to start by working with only 1% of the dataset, around 12k reviews. For this size, an operation such as count vectorization takes around two minutes, which is about the maximum one could have patience with when building an analytics pipeline. Once I have built a robust pipeline that provides meaningful insights, I will scale up my analysis to 10% to increase my analytical power, with the final intention of processing the entire dataset using cluster computing on a platform such as Databricks.

The purpose of this project is to use high throughput natural language processing (NLP) on Apache Spark to investigate the features that are associated with one-star versus five-star reviews. First, I loaded the entire reviews .json file into Spark. I combined the reviews .json file with the businesses .json file (which contains business categories), and filtered by restaurant, in order to limit the scope of my analysis. I filtered by one and five-star reviews, in order to have greater contrast in my labels. Because there are many more five-star than one-star reviews, I limited the number of five-star reviews to the number of one-star reviews in order to avoid class imbalance.

Next, I built my NLP preprocessing pipeline using Spark NLP, which involved tokenization, normalization, lemmatization, stopword removal, and bigram generation. I created a term frequency-inverse document frequency (tf-idf) vector upon which I would build my models.

Modeling started out with an 80-20 train-test split. The first model I attempted was a Latent Dirichlet allocation (LDA) model using Spark MLlib. However, this model was not very successful because the topics it generated referred more to the general features of restaurants (Asian, upscale, etc.) rather than the features associated with one versus five-star reviews. Second, I attempted a logistic regression model. However, this model did not align with the purpose of the project, which was not to predict one versus five-star reviews based on review text, but to understand what makes a one versus five-star review. Finally, I tried a random forest model, which is what I am currently working on. This appears to be the most appropriate approach because you can extract feature importances that are relative to their efficacy in classifying one versus five-star reviews.

I am still working on various challenges at all stages of this analytics pipeline. First, I am trying different approaches regarding using only unigrams, unigrams + bigrams, only bigrams, or larger n-grams. Using only unigrams tends to generate features that are too general and obvious such as “delicious,” while using bigrams or larger n-grams fails to capture important features due to the highly combinatorial nature of language. One way to avoid this is to use part of speech (POS)-based chunking to identify meaningful bigrams and reduce noise in the dataset, which I may have to eventually implement. Another challenge involves the minimum and maximum document frequency (minDF and maxDF) parameters of count vectorization (i.e., the minimum and maximum number of documents a token has to appear in in order to be considered). If the minDF is too low, features will be too esoteric, and if the max DF is too high, features will be too general. I am currently working on playing with these parameters to generate the most insightful and salient results possible.

Overall, there is still much work to do, but if successful, this model should output meaningful, insightful, and salient features that could aid restaurant owners in making changes to avoid getting one-star reviews and acquire more five-star reviews.

References

Chambers, B., & Zaharia, M. (2018). Spark: The Definitive Guide.

Damji, J. S., Wenig, B., Das, T., & Lee, D. (2020). Learning Spark: Lightning-Fast Data Analytics.

Ghosh, S., & Gunning, D. (2019). Natural Language Processing Fundamentals.

Lane, H., Howard, C., & Hapke, H. M. (2019). Natural Language Processing in Action: Understanding, analyzing, and generating text with Python.

Sarkar, D. (2019). Text Analytics with Python: A Practitioner’s Guide to Natural Language Processing.

Talby, David. “Introducing the Natural Language Processing Library for Apache Spark - The Databricks Blog.” Databricks, 19 Oct. 2017, databricks.com/blog/2017/10/19/introducing-natural-language-processing-library-apache-spark.html.