Scalable Real-time Streaming ETL for Spam Detection

Brian White, Shih-Chieh (Jack) Chen, Jiuwei Wang

Abstract

In this project we focus primarily on designing a scalable system to support near-online learning applications involving high-throughput, computationally expensive data stream processing in near real time. Secondarily we focus on building a spam detector model that utilises this framework. We make use of state-of-the-art components from the Apache ecosystem to achieve highly scalable and fault tolerant data ingestion, streaming, feature engineering, storage, and model training. In addition to the end-to-end spam detection system, we also consider practical deployment and development strategies for building such cloud-destined, distributed applications.

INTRODUCTION

Traditional machine learning models employ batch learning algorithms, in which the process of training does not lend itself to support incremental updates when new data come available -- rather, the entire system must be trained from scratch using the full dataset to utilise any new training instances. As the size of training data sets continues to grow, along with an increase in demand for compute-intensive applications, training time can quickly become a significant challenge.

In the case of machine learning applications that explore patterns in sequential data, it is often the case for most relevant patterns to be observed in recent instances. For example, in the field of fraud detection, patterns in fraudulent transactions are often short-lived, as fraudsters are continuously changing their techniques to fool the fraud detection system. Other examples include spam detection, and financial modelling.

In this project, we design and implement infrastructure to accommodate such applications using open-source big data technologies from the Apache ecosystem, namely, Kafka, Spark Streaming, Spark ML, and Cassandra. Further, we use Docker to containerize large parts of our system in order to simplify the process of horizontal-scaling and deploying to the cloud. Finally, we develop an end-to-end spam detection machine learning application which is retrained periodically using only recent training examples to demonstrate the potential of our infrastructure in real-world problems.

HIGH LEVEL SYSTEM ARCHITECTURE

At a high level, our system can be broken down into three parts:

1. Kafka producers and broker
2. Kafka consumer and ETL
3. Spam detector

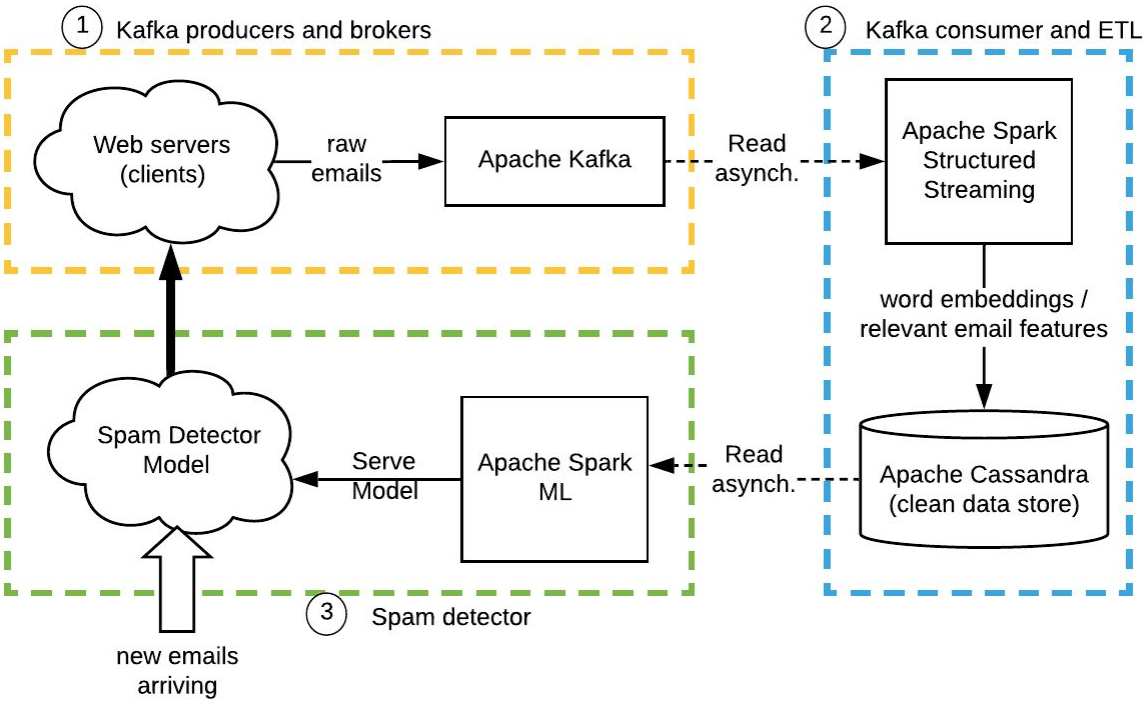

Figure 1: High level system architecture for end-to-end spam detector

BACKGROUND AND MOTIVATION

Gaining inspiration from recent works in the field of fraud detection[1], at the core of our infrastructure is a (near) real time streaming ETL using Kafka, Spark Streaming, and Cassandra. We imagine the process of spam detection to work approximately as follows, and design our system with this process in mind:

1. Incoming mail is classified as spam / ham by an existing spam detector model, which places the new mail in the appropriate folder of the recipient's email client.

2. The recipient may re-classify the mail after it has been initially classified by the spam detector, (assuming of course that the spam detector may have made mistakes). After a certain time has elapsed, the mail is assumed to be more accurately classified than when it first arrived, and thus is treated as an accurately labeled training instance, which we then send off to Kafka to be analysed asynchronously.

3. Kafka will retain these messages for a period of time, in which a consumer process, (a Spark Structured Streaming application in our case), reads a batch of messages from Kafka, extracts relevant information, and sends this information (extracted features) to Cassandra to be retained for a longer time (weeks to months) and processed by other applications down stream.

4. When enough new data has arrived in Cassandra, the spam classifier is retrained, (using all data available in Cassandra -- that is, data that is relatively new), and is deployed to replace the previous model and thus focus more closely on recent trends in spam vs. ham mail.

REAL TIME DATA FEED WITH KAFKA

At the backbone of our system is Apache Kafka, a fault-tolerant, distributed event streaming platform. Kafka supports multiple producers and consumers, making it a suitable tool for our spam detection system -- wherein, many producers correspond to many email clients that would be present if this tool were to be deployed to production, (albeit, in our system we have only experimented with a few producers at a time).

The Kafka system can be most easily explained diagramatically:

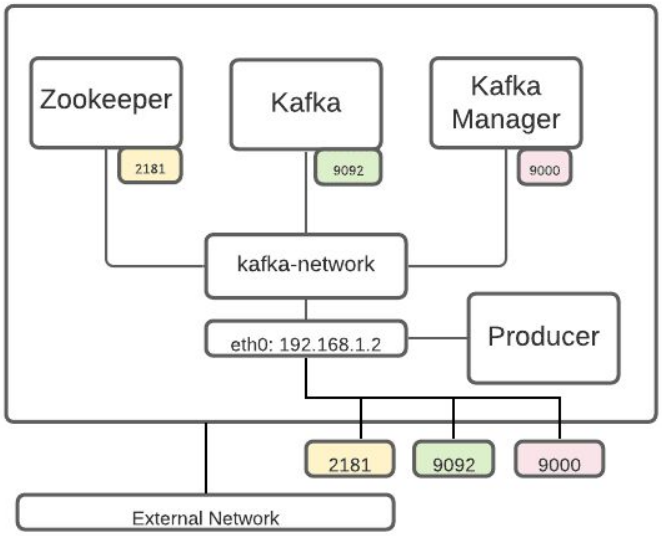

Figure 2: Kafka cluster connectivity

In the image above, the Kafka cluster is contained in it's own network, isolated from the rest of the system as would be the case in the practice. This network structure is controlled via a set of environment variables specified in the docker-compose.yml file in our code repository [3], together with various docker commands and options, (also documented in our repository in the README.md file). With the help of docker and docker-compose commands we can easily scale the system up or down by creating more Kafka brokers, or Zookeeper instances. By keeping our kafka cluster fully containerised, we further benefit from the minimal changes required to our workflow when we wish to deploy in a production environment, such as on AWS, (see more below).

FEATURE ENGINEERING WITH SPARK

For our ETL component, we chose Spark for it's parallel computational abilities and its ability to seamlessly communicate with Kafka and Cassandra. The flexibility of Spark allows the system to scale with Kafka as incoming data increases. The feature we selected for our model was based on practices from Natural Language Processing systems. Firstly, identifying information is extracted (sender, recipient, date received, and email subject). While these variables are not used to train the spam filter, we considered them relevant information for other designs of the spam filter and are important information when maintaining a database for emails. The main purpose of the ETL process is to extract relevant information from the emails contents. This proved to be a challenge since Spark's ability to process text is limited to regex expressions, which are not effective in removing markup languages. Some computational speed was sacrificed in exchange for cleaner output by incorporating available python libraries as user defined functions. Additional features were extracted by assigning aliases to common patterns that would otherwise complicate further analyses. For example, numbers are replaced with the corresponding text "numbers" and URLs are replaced with the text "httpaddr." Verb conjugations and letter case are simplified to word stems and lower case respectively. Again, user defined functions were implemented due to the limitations of Spark. The text is then tokenized and all relevant information is stored in Cassandra.

STORAGE WITH CASSANDRA

Given the nature of our system, many write processes are expected, as emails are constantly being processed in real time and written into Cassandra. On the other hand, reading processes only occur during retraining. This couples well with Cassandra's strength in write processes and it's weakness in read processes. In addition, we expect our data size will expand quickly given the prevalence of emails. Suitable to our needs, Cassandra can be easily scaled to more nodes and storage space. While Cassandra cannot achieve perfect consistency, it can achieve eventual consistency given time. This weakness is not an issue for our system. The gaps between each model retraining is expected to be days and weeks apart. That is plenty of time for any inconsistencies to be resolved. While some records could be missing due to inconsistencies during retraining, those records can be disregarded as the data loss relative to the entire database will be small. Those data will eventually be incorporated into the model in the next retraining.

TRAINING A CLASSIFIER WITH SPARK ML

Given the size of our data, we decided to implement a technique called Feature Hashing. In Feature Hashing, a word is associated with an index value and each email is transformed into a vector containing a word's index and a number indicating its occurrence. For our model, we chose the Linear Support Vector Classification model. When training, we split our data into a 3:1 ratio for training and validation data respectively. The final output produces an accuracy estimate of the validation data.

DEVELOPMENT AND DEPLOYMENT LESSONS

Because all of the technologies used in our project are open-source, and since significant components of our system are already fully containerised, we have kept our system as flexible as possible. We can quite easily deploy our system to a cloud service provider such as AWS, yet still avoid committing to any vendor lock-in. By utilising the recently developed Docker Compose CLI [4], we can develop and test our system using native Docker commands to run our application in Amazon EC2 Container Service (ECS). In this way, we keep our system completely free from an AWS dependency, yet still enable developers to subscribe to AWS resources for testing the system in a production development.

PROBLEMS / DIFFICULTIES

1. Initially, our team hoped to build the entire system on Amazon Web Services. After several test runs it became apparent that AWS has a difficult learning environment and experimenting with AWS can quickly get expensive. In the end, we chose to build Kafka in docker, and install spark and Cassandra on local and school systems to complete the entire system construction.

2. When designing the ETL steps, it became apparent that many of the functions in Spark are not equipped to clean our data. While we were hoping to work only with Spark functions, we finally decided to use UDFs for cleaner and more effective code.

3. Managing the data files to be downloaded in a script and properly stored in the "data directory".

4. Developing Kafka in Docker had its own issues. Many of which involve tweaks to get Kafka, which is inside the Docker, to Spark, which is outside the Docker.

PROJECT SUMMARY

1.	Getting the data: Acquiring/gathering/downloading *-- See producer/utils.py*	2
2.	ETL: Extract-Transform-Load work and cleaning the data set *-- Extract from Kafka, transform in spark, load into Cassandra. See consumer/SparkETL.py*	4
3.	Problem: Work on defining the problem itself and motivation for the analysis. *-- See Background and Motivation section of this report*	2
4.	Algorithmic work: Work on the algorithms needed to work with the data, including integrating data mining and machine learning techniques. *-- Create a spark ml pipeline which uses Support Vector Classification algorithms.*	2

5.	Bigness/parallelization: Efficiency of the analysis on a cluster, and scalability to larger data sets.	5
6.	UI: User interface to the results, possibly including web or data exploration frontends.	1
7.	Visualization: Visualization of analysis results.	0
8.	Technologies: New technologies learned as part of doing the project. *-- Utilizing Kafka and Docker*	4

REFERENCES

1. Carcillo F, Dal Pozzolo A, Le Borgne Y-A, Caelen O, Mazzer Y, Bontempi G. SCARFF: a scalable framework for streaming credit card fraud detection with spark. Inf Fusion. 2018;41:182–94
2. https://github.com/wurstmeister/kafka-docker
3. https://csil-git1.cs.surrey.sfu.ca/732-project/bigspamproject
4. https://docs.docker.com/engine/context/ecs-integration