University Of Crete
Department of Computer Science

Feature Extraction for Multi-layer Graph Link Prediction

Dimitrios Andreas Vogiatzidakis

Supervisor: Polyvios Pratikakis

A thesis submitted in partial fulfilment of the University's requirements
for the degree of Bachelor of Science

July 5, 2020

Abstract

Data sets can naturally be represented as graph where nodes represent instances and links represent relationships between those instances.A fundamental issue with these types of data is that links may incorrectly exist between unrelated nodes or that links may be missing between two related nodes. The goal of link prediction is to predict those irregularities between the nodes of the graph and also possible future links.
This thesis tries to reproduce the feature extraction algorithm of TwitterMancer[1] using the Apache Flink framework. Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams, designed to run in all common cluster environments and perform computations at in-memory speed and at any scale. We will use this framework to speedup the process of feature extraction and as a result, analyze larger scale graphs.
Using a data set of twitter interactions (follow, retweet, reply, quote) we find that we can achieve great speedup as we increase the number of cores of our cluster. That means the project can be scaled by simply adding more machines to the cluster. Substantially decreasing the time required to extract features from a data set , will result to faster and better results on link prediction.

Contents

List of Figures

Chapter 1

Introduction

Traditionally, programs have been written for serial computation. That means a single problem is broken into smaller instructions, those instructions are executed sequentailly, one at a time. Parallel computation is in the simplest way, the simultaneous execution of these instructions to compute the problem. It was introduced as way to model scientific problems, such as meteorology. This led to the design of parallel hardware and software which allowed the use of larger datasets. In this thesis we are trying to replicate the work of Twittermancer[1] using parallel computation. This will allow us to use larger datasets and maybe improve the accuracy of the results. Our goal is to create a fast and reliable way to extract features out of a given dataset.

1.1 Challenges

One of our main concerns was regarding the DRAM as our algorithm will produce a cross-product of N nodes, that will later be used to extract the features. Luckily, Flink is implemented in a way to make full use of the available memory without ever exceeding it and causing errors.

The original goal was to use the extracted features and perform a logistic regression. However after version 1.8 Apache Flink no longer supports the Machine Learning library and is currently in early stages of being reimplemented. Our solution was to extract the features, and create a new dataset containing those features, that could be used by another program to perfome the Logistic Regression. This created another issue with our filesystem memory as the output of our algorithm is a .csv document containing all features for each pair of nodes(edge). Thus we have a file of "N*N*Number of Features" records taking a lot of space on our hard drives. We partially solved this for testing purposes using datasets that will output the maximum available size and not more.

Chapter 2

Background

2.1 Theory Overview

In this thesis we are trying to extract features from a given dataset in order to perform a Logistic Regression and predict missing or future links.

2.1.1 Feature Extraction

Full definition taken from wikipedia (en.wikipedia.org/wiki/Feature_extraction):

> In machine learning, pattern recognition and in image processing, feature extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. Feature extraction is related to dimensionality reduction.
>
> When the input data to an algorithm is too large to be processed and it is suspected to be redundant (e.g. the same measurement in both feet and meters, or the repetitiveness of images presented as pixels), then it can be transformed into a reduced set of features (also named a feature vector). Determining a subset of the initial features is called feature selection.[1] The selected features are expected to contain the relevant information from the input data, so that the desired task can be performed by using this reduced representation instead of the complete initial data.

2.1.2 Logistic Regression

Full definition taken from Research methods inhuman skeletal biology[2]:

> Logistic regression (LR) is a statistical method similar to linear regression since LR finds an equation that predicts an outcome for a binary variable, Y, from one or more response variables, X. However, unlike linear regression the response

variables can be categorical or continuous, as the model does not strictly require continuous data. To predict group membership, LR uses the log odds ratio rather than probabilities and an iterative maximum likelihood method rather than a least squares to fit the final model. This means the researcher has more freedom when using LR and the method may be more appropriate for nonnormally distributed data or when the samples have unequal covariance matrices. Logistic regression assumes independence among variables, which is not always met in morphoscopic datasets. However, as is often the case, the applicability of the method (and how well it works, e.g., the classification error) often trumps statistical assumptions. One drawback of LR is that the method cannot produce typicality probabilities (useful for forensic casework), but these values may be substituted with nonparametric methods such as ranked probabilities and ranked interindividual similarity measures.

2.2 Tools Overview

The project was created using Java and Apache-Flink. We build the project using Apache Maven.

2.2.1 Apache Flink

All information were taken from Apache Flink Website:

> Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams, designed to run in all common cluster environments and perform computations at in-memory speed and at any scale.
>
> - Unbounded streams have a start but no defined end. They must be continuously processed and it is not possible to wait for all input data to arrive.
> - Bounded streams have a defined start and end. Bounded streams can be processed by ingesting all data before performing any computations. Processing of bounded streams is also known as batch processing.
>
> Apache Flink excels at processing unbounded and bounded data sets. Precise control of time and state enable Flink's runtime to run any kind of application on unbounded streams. Bounded streams are internally processed by algorithms and data structures that are specifically designed for fixed sized data sets, yielding excellent performance.
>
> Apache Flink is a distributed system and requires compute resources in order to execute applications. Flink integrates with all common cluster resource managers such as Hadoop YARN, Apache Mesos, and Kubernetes but can also be setup to run as a stand-alone cluster.It is designed to work well each of the previously listed resource managers. This is achieved by resource-manager-specific

deployment modes that allow Flink to interact with each resource manager in its idiomatic way.

When deploying a Flink application, Flink automatically identifies the required resources based on the application's configured parallelism and requests them from the resource manager. In case of a failure, Flink replaces the failed container by requesting new resources. All communication to submit or control an application happens via REST calls. This eases the integration of Flink in many environments.

Flink is designed to run stateful streaming applications at any scale. Applications are parallelized into possibly thousands of tasks that are distributed and concurrently executed in a cluster. Therefore, an application can leverage virtually unlimited amounts of CPUs, main memory, disk and network IO. Moreover, Flink easily maintains very large application state. Its asynchronous and incremental checkpointing algorithm ensures minimal impact on processing latencies while guaranteeing exactly-once state consistency.

Stateful Flink applications are optimized for local state access. Task state is always maintained in memory or, if the state size exceeds the available memory, in access-efficient on-disk data structures. Hence, tasks perform all computations by accessing local, often in-memory, state yielding very low processing latencies. Flink guarantees exactly-once state consistency in case of failures by periodically and asynchronously checkpointing the local state to durable storage.

2.2.2 Apache Maven

All information were taken from Apache Maven website:

Maven began as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects, each with their own Ant build files, that were all slightly different. JARs were checked into CVS. They needed a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information, and a way to share JARs across several projects. The result is a tool that can now be used for building and managing any Java-based project.

Maven's primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal, Maven deals with several areas of concern:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Encouraging better development practices

While using Maven doesn't eliminate the need to know about the underlying mechanisms, Maven does shield developers from many details. It builds a project

using its project object model (POM) and a set of plugins. Maven provides useful project information that is in part taken from your POM and in part generated from your project's sources. For example, Maven can provide:

- Change log created directly from source control
- Cross referenced sources
- Mailing lists managed by the project
- Dependencies used by the project
- Unit test reports including coverage

Third party code analysis products also provide Maven plugins that add their reports to the standard information given by Maven.

Chapter 3

Methodology

3.1 Requirements

In order to run the project, there are some requirements:

- A working Java 8 or 11 installation.
- Have the flinkmancer.jar .
- Download Apache Flink 1.10.1 for Scala 2.12

After the downloaded archive is unpacked, flink has to be configured for the standalone cluster. The "/conf" directory has 2 important files.

- flink-conf.yaml:

  In this file we set:
  "jobmanager.rpc.address" to be the IP of Job Manager.
  "taskmanager.memory.flink.size" to be the highest available DRAM.
  "taskmanager.memory.off-heap.size" to at least 1024m.
  "taskmanager.numberOfTaskSlots" to the number of cores each machine has.
  "taskmanager.memory.network.min" and "taskmanager.memory.network.max" in a way that flink.size * network.fraction (0.1) is between min and max.

- slaves:

  In this file we add the id of each machine that will run a task manager.

Now we can start our cluster and run the project

- " ./flink-1.10.0/bin/start-cluster.sh "
- " ./flink-1.10.0/bin/flink run flinkmancer.jar --cores [number of total cores] --path [path to dataset] --outpath [path to output file inclunding the name]"

The datasets needs to be in ./data/layer for each different layer used.
In our test runs we used 4 different layers, “follow”, ”reply”, ”quote”, ”retweet”. After the program is executed , it produces a Features.csv file, containing 100 Features about each Edge of Vectors (each Vector being a user) and their common neighbors.

3.2 Design

We wanted to create a dataset of every node paired with a list of every neighbor for every layer examined in the process. We achieved that, by reading each layers data, creating 2 adjacent lists for every layer, one for incoming neighbors and one for outgoing. Then we used outer join to combine them and have an 'empty' slot for every missing link.

For example: Imagine we had Node 1 , with incoming nodes 2,3 and outgoing 3,4. What outer join does for us is that our result is a Tuple of 2 lists ([2,3,[]],[[],3,4]) instead of ([2,3][3,4]).

We use the same technique for every Tuple we created on the previous step and we combine them in a final Tuple of 8 lists (4 layers in this process). Now we have a Dataset containting a Tuple of (Id,Tuple8(list of neighbors for each layer)).

Now we create a cross product of this dataset with itself. This will create every possible edge given our known ids. For each of these edges, we are going to extract the features we need. We take each edge and perform and intersection on every neighbor list to find their common neighbors and count them. Those features are explained in details in the TwitterMancer[1] paper.

In our tests we have a total of 100 features for each Edge, so we produce an output of 102 numbers (2 node ides and 100 features) for each Edge in the cross product we created.

The diagram of our project is shown in Figure 1.

3.3 Implementation

This is a Maven Project written in Java using Apache Flink.

It contains 3 .java files:

- Flinkmancer.java
- Features.java
- Node.java

Flinkmancer.java is main class of the project.It sets up the flink environement, reads the datasets for each layer and creates the adjacent lists of neighbors which then combines using outer join function. These lists are transformed into Nodes. Those nodes are then used to create a cross product with themselves. using .cross function

provided by flink, and stored in a Dataset. we apply a Feat() function to every edge in our cross product and we recieve a final Tuple of 2, (ids,features).

Node.java is simply the class that implements the Node objects, and contains ways to access each object variables such us setId(), getId(), getSets() as well as the class constructors.

Features.java contains the feature extraction function Feat(). This function uses the dataset of all Nodes created in Flinkmancer.java to extract 100 features for each Node Edge. It returns an array of 102 columns, first 2 being the ID of nodes , and rest 100 the features in form of Tuple2 .

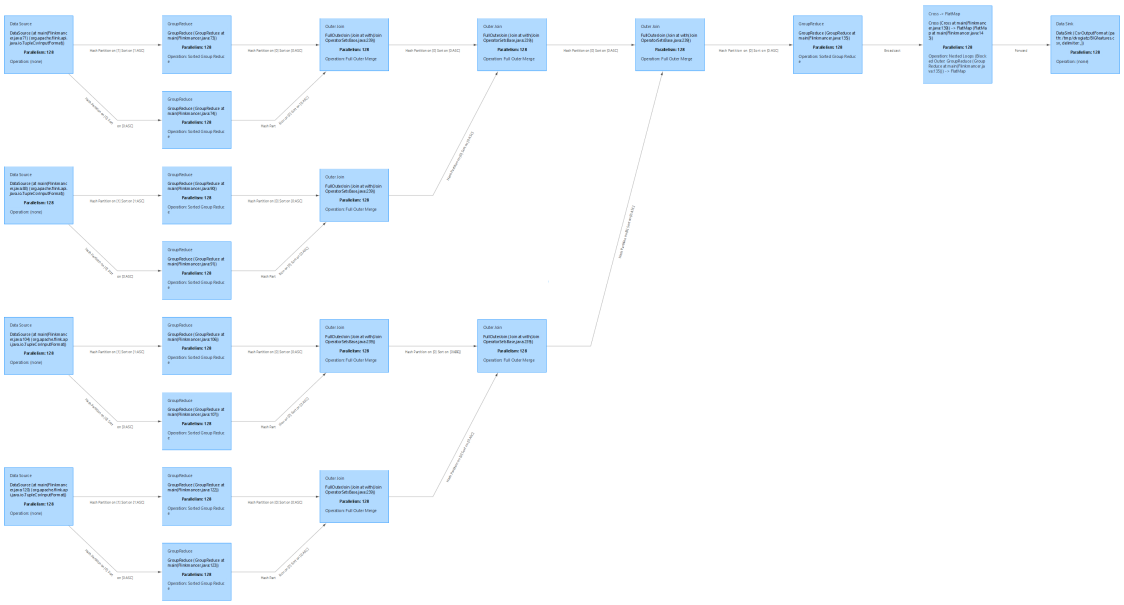

Figure 3.1: *Every process that takes place in the project*

Chapter 4

Results

4.1 Experiment 1

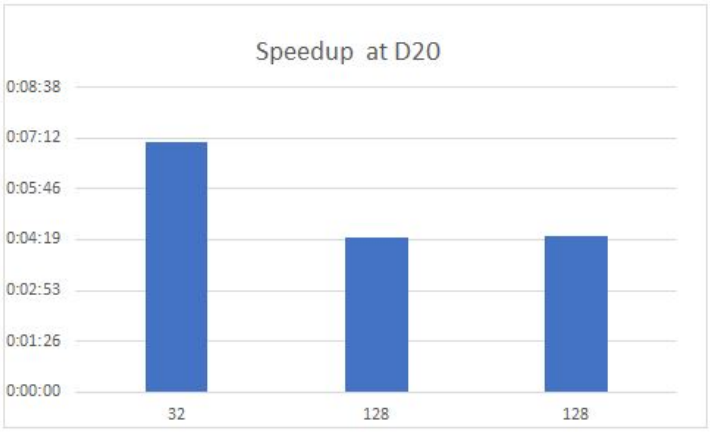

Figure 4.1: *Sppedup at D20 dataset from 32 to 128 cores*

4.2 Experiment 2

Chapter 5

Conclusions and Future Work

In conclusion we found out that the immediate process of the features extracted is required, because of the limitation we had in disk space. Since the project creates a cross product, the output size has quadratic growth. So in our tests we run quickly out of space as the increased the input size. If in the future the Machine learning library is reimplemented in Apache Flink, then we could do our computation inside the same framework.

One suggestion we have for future work is to use Apache Kafka or some alternative as a data sink and redirect the features to an Apache Spark project that could perform the logistic regression.

Bibliography

[1] K. Sotiropoulos, J. W. Byers, P. Pratikakis, and C. E. Tsourakakis. Twittermancer: Predictinginteractions on twitter accurately.arXiv preprint arXiv:1904.11119, 2019.
[2] DiGangi EA, Moore MK, editors. 2012. Research methods inhuman skeletal biology. Oxford: Academic Press.