DIT178 2022-2023 Project Report
Αναστάσιος Κοτρώνης
itp22104

1 Γενικά

Στην παρούσα εργασία υλοποιούνται τρεις μετρικές για την πρόβλεψη ακμών σε μη κατευθυνόμενα γραφήματα, με οδηγό την περιγραφή του προβλήματος στη δημοσίευση:

> *"The link-prediction problem for social networks."* - Liben-Nowell, David, and Jon Kleinberg. Journal of the American society for information science and technology 58.7 (2007): 1019-1031.

Η υλοποίηση γίνεται με τη βιβλιοθήκη **pyspark** της γλώσσας **python**.

2 Περιγραφή

Το πρόγραμμα δέχεται ως είσοδο ζεύγη ακεραίων, κάθε ένα από τα οποία αναπαριστά την ακμή μεταξύ δυο κόμβων σε ένα μη κατευθυνόμενο γράφημα. Για κάθε δυνατή μη κατευθυνόμενη ακμή από κόμβους η οποία **δεν** περιλαμβάνεται στα αρχικά ζεύγη, υπολογίζει τρεις μετρικές κάθε μια από τις οποίες εκτιμά κατά πόσο είναι πιθανό η ακμή να υπάρξει με βάση τις υπάρχουσες ακμές που έλαβε ως είσοδο.

Οι μετρικές που υπολογίζονται ορίζονται ως εξής: Για δυο κόμβους u, v, συμβολίζοντας με $\Gamma(v)$ το σύνολο των γειτόνων του κόμβου v:

1. **Common Neighbors (cn)**: $score(u, v) = |\Gamma(u) \cap \Gamma(v)|$,
2. **Jaccard Coefficient (jc)**: $score(u, v) = \dfrac{|\Gamma(u) \cap \Gamma(v)|}{|\Gamma(u) \cup \Gamma(v)|}$,
3. **Adamic/Adar (aa)**: $score(u, v) = \displaystyle\sum_{z \in \Gamma(u) \cap \Gamma(v)} \frac{1}{\log |\Gamma(z)|}$.

3 Υλοποίηση

3.1 Βοηθητικές Συναρτήσεις

Με την παρακάτω συνάρτηση η κάθε γραμμή του αρχείου εισόδου αντιστοιχίζεται στο -1 αν είναι σχόλιο (ξεκινάει από #), ή σε ζεύγος ακεραίων, αφού πρώτα έχει μετατραπεί σε ζεύγος κόμβων σε περίπτωση ακμής μεταξύ ενός κόμβου:

```
def parse_line(edge):
    if not edge.startswith('#'):
        nodes = edge.split()
        if len(nodes) == 1:
            nodes = 2 * nodes
        return tuple(map(int, nodes))
    return -1
```

Εδω τα x, y είναι λίστες με γειτονικούς κόμβους ενός κόμβου και ορίζουμε συνάρτηση που ενώνει τα στοιχεία της πρώτης με αυτά της δεύτερης που δεν ανήκουν ήδη στην πρώτη:

```
def agg_neighbors(x, y):
    x.extend([n for n in y if n not in x])
    return x
```

Εδω το στοιχείο εισόδου είναι της μορφής $(n, [nb_1, nb_2, \ldots])$, όπου $nb_1, nb_2, \ldots$ είναι οι γειτονικοί κόμβοι του κόμβου n. Επιστρέφουμε στην ίδια μορφή το στοιχείο $(n, [nnb_1, nnb_2, \ldots])$, όπου $nnb_1, nnb_2, \ldots$ είναι οι κόμβοι του γραφήματος οι οποίοι **δεν** είναι γειτονικοί του n και $nnb_i > n$. Το τελευταίο διότι το γράφημα είναι μη κατευθυνόμενο οπότε οι ακμές (n_1, n_2) και (n_2, n_1) είναι οι ίδιες και αγνοούμε τις ακμές μεταξύ ενός ίδιου κόμβου για την πρόβλεψη. Η μεταβλητή nodes είναι broadcasted (βλέπε παρακάτω) και έχει όλους τους κόμβους:

```
def nodes_with_not_neighbors(line):
    base_node, neighbors = line
    not_neighbors = [node for node in nodes.value
        if node not in neighbors and node > base_node]
    return base_node, not_neighbors
```

Στην παρακάτω συνάρτηση το στοιχείο εισόδου είναι ακμή της μορφής (n_1, n_2). Βρίσκουμε τους γειτονικούς κόμβους για τους n_1, n_2 απο τη μεταβλητή nodes_with_neighbors_brd (broadcasted) και υπολογίζουμε την αντίστοιχη μετρική σύμφωνα με τους παραπάνω τύπους ανάλογα με το input που έχει δώσει ο χρήστης κατά την εκτέλεση του προγράμματος (***cn***=*Common Neighbors*, ***jc***=*Jaccard Coefficient*, ***aa***=*Adamic/Adar*):

```
def calculate_scores(nodes):
    node_1, node_2 = nodes
    neighbors_1, neighbors_2 = map(
        set,
        [nodes_with_neighbors_brd.value[node_1],
        nodes_with_neighbors_brd.value[node_2]]
    )
    common_neighbors = neighbors_1.intersection(neighbors_2)
    if args.score_metric == 'jc':
        score = len(common_neighbors) /
        (len(neighbors_1) +
         len(neighbors_2) -
         len(common_neighbors))
    elif args.score_metric == 'aa':
        score = sum(
            [1 /
             math.log(
                 len(nodes_with_neighbors_brd.value[nd])
             )
             for nd in common_neighbors]
        )
```

```
        else:
            score = len(common_neighbors)
        return (node_1, node_2), score
```

3.2 Κύριο Μέρος Κώδικα

Ορίζουμε το SparkContext και το configuration του με master την είσοδο του χρήστη κατά την εκτέλεση του προγράμματος. Default τιμή "local[*]"

```
conf = SparkConf()\
    .setMaster(args.master)\
    .setAppName("DIT178")
sc = SparkContext(conf=conf)
```

Διαβάζουμε το αρχείο κάνουμε map την parse_line και κρατάμε τις γραμμές που μας ενδιαφέρουν

```
edges = sc.textFile(data_file)\
    .map(parse_line)\
    .filter(lambda x: x != -1)
```

Με τον παρακάτω κώδικα:

1. Για κάθε ζευγάρι κόμβων από το edges RDD δημιουργούμε μια νέα γραμμή με τους κόμβους αντεστραμμένους ώστε να έχουμε όλους τους κόμβους σαν κλειδί σε γραμμή
2. Το δεύτερο κόμβο κάθε γραμμής τον κάνουμε λίστα ώστε να
3. εφαρμόσουμε τη μέθοδο reduceByKey με τη βοηθητική συνάρτηση agg_neighbors για να έχουμε ένα RDD με κλειδιά όλους τους κόμβους και τιμές μία λίστα με τους γειτονικούς κόμβους τους.
4. Κάνουμε cache το RDD nodes_with_neighbors ώστε να αποφύγουμε επαναϋπολογισμό του

```
nodes_with_neighbors = edges\
    .flatMap(lambda edge:[edge, edge[::-1]])\
    .map(lambda edge:[edge[0], [edge[1]]])\
    .reduceByKey(lambda x, y: agg_neighbors(x, y))\
    .cache()
```

Κάνουμε broadcast το RDD nodes_with_neighbors ως λεξικό (nodes_with_neighbors_brd) και τους κόμβους του γραφήματος ως λίστα (nodes). Οι μεταβλητές αυτές στα αρχεία δοκιμής του κώδικα (18772 και 23133 κόμβοι αντίστοιχα το κάθε γράφημα) καταλαμβάνουν μικρό χώρο (0.56mb και 0.16mb αντίστοιχα για το πρώτο αρχείο και 1.25mb και 0.19mb αντίστοιχα για το δεύτερο), οπότε είναι συμφέρον να τις χρησιμοποιήσουμε με αυτόν τον τρόπο προκειμένου να γλυτώσουμε ενδεχόμενο join του RDD με τις προς υπολογισμό ακμές (≈ 176mil και ≈ 267mil για το πρώτο και δεύτερο αρχείο) με το RDD nodes_with_neighbors.

```
nodes_with_neighbors_brd = sc.broadcast(
    nodes_with_neighbors.collectAsMap()
)
nodes = sc.broadcast(
    list(nodes_with_neighbors_brd.value.keys())
)
```

Στη συνέχεια, ξεκινώντας από το cached RDD nodes_with_neighbors:

1. Χρησιμοποιώντας τη συνάρτηση nodes_with_not_neighbors, φτιάχνουμε για κάθε κόμβο μια λίστα με τους κόμβους οι οποίοι **δεν** είναι γειτονικοί του,
2. Μέσω της flatMapValues φτιάχνουμε ένα RDD που έχει γραμμές όλες τις προς υπολογισμό ακμές,
3. Με τη calculate_scores όπως ορίστηκε παραπάνω υπολογίζουμε το score της κάθε ακμής.

```
scored_edges = nodes_with_neighbors\
    .map(nodes_with_not_neighbors)\
    .flatMapValues(lambda x:x)\
    .map(calculate_scores)
```

Με το action takeOrdered γίνονται οι παραπάνω υπολογισμοί και επιστρέφονται οι ακμές με τα μεγαλύτερα scores, όσες στο πλήθος ζήτησε ο χρήστης κατά την εκτέλεση του προγράμματος:

```
selected_results = scored_edges\
    .takeOrdered(args.edges_number, key=lambda x: -x[-1])
```

Τέλος τυπώνονται τα αποτελέσματα στην κονσόλα και κλείνουμε το sparkContext

```
for result in selected_results:
    ...

sc.stop()
```

4 Testing

Αρχεία ελέγχου του προγράμματος

1. `https://snap.stanford.edu/data/ca-AstroPh.html`
2. `https://snap.stanford.edu/data/ca-CondMat.html`

4.1 Environment Setup

Δημιουργία και ενεργοποίηση virtual environment, εγκατάσταση dependencies στο virtual environment

```
virtualenv -p python3 .venv
source .venv/bin/activate
pip3 install -r requirements.txt
```

4.2 Εκτέλεση

```
python main.py \
    --input ca-AstroPh.txt.gz \
    --edges-num 30 \
    --score-metric cn \
    --master "local[*]"
```

1. Τα input αρχεία πρέπει να είναι στο ίδιο folder με το αρχείο main.py.
2. edges – num → Ο αριθμός των top scored ακμών που θα επιστρέψει το πρόγραμμα
3. score – metric →
   - cn → (Common Neighbors)
   - jc → (Jaccard Coefficient)
   - aa → (Adamic/Adar)