2023-04-05

Πανεπιστήμιο Πειραιώς

Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης

Τελική Εργασία για το Μάθημα

Ανάλυση Κοινωνικών Δικτύων

των

Αποστόλου Αθανάσιου (mpsp2203@unipi.gr)

Μπιρμπάκου Γεώργιου (mpsp2220@unipi.gr)

Θέμα: Time-Aware Network Centrality Measures & Link Prediction

Table of Contents

Εισαγωγή

Στα πλαίσια της τελικής εργασίας του μαθήματος “Ανάλυση Κοινωνικών Δικτύων” αναπτύξαμε κώδικα όπου διαβάζει το “sx-stackoverflow” Dataset (το οποίο βρίσκεται στην διεύθυνση https://snap.stanford.edu/data/sx-stackoverflow.html), αναλύει τα δεδομένα του και εξάγει τα επιθυμητά αποτελέσματα.

Ο κώδικας έχει γραφεί σε γλώσσα Python. Χρησιμοποιούμε το poetry package manager για ευκολότερη διαχείριση των dependencies. Ο κώδικας και αυτή η αναφορά βρίσκονται επίσης στο git repository https://github.com/ThanosApostolou/aics-social-networks-analysis. Για την εκτέλεση του κώδικα χρειάζεται η εγκατάσταση του εργαλείου “poetry” (είτε μέσω pip ή κάποιου άλλου τρόπου) και έπειτα η εκτέλεση των παρακάτω εντολών από το root του repository:

- poetry install
- poetry run python social_networks_analysis/main.py

Για να τρέξει ο κώδικας χρειάζεται ένα μηχάνημα με τουλάχιστον 16GB RAM καθώς οι απαιτήσεις για την εξαγωγή των δεδομένων των δικτύων καθώς και η εκπαίδευση των νευρωνικών δικτύων είναι απαιτητικές.

Προ-επεξεργασία Δεδομένων

Ο κώδικας της εργασίας μας αρχίζει από το αρχείο “main.py” και είναι χωρισμένος σε 4 βασικές συναρτήσεις. Τις “run_intro()”, “run_part1()”, “run_part2()” και “run_part3()”.

 # main.py

```
def main():
    # run intro
    run_intro_output: RunIntroOutput = run_intro()
    # run part1
    run_part1(RunPart1Input(run_intro_output.sx_df, run_intro_output.t_min,
run_intro_output.t_max,
              run_intro_output.DT, run_intro_output.dt,
run_intro_output.time_spans))
    # run part2
    run_part2(RunPart2Input(run_intro_output.sx_df, run_intro_output.t_min,
run_intro_output.t_max,
                            run_intro_output.DT, run_intro_output.dt,
run_intro_output.time_spans))
    # run part3
    run_part3(RunPart3Input(run_intro_output.sx_df, run_intro_output.t_min,
run_intro_output.t_max,
                            run_intro_output.DT, run_intro_output.dt,
run_intro_output.time_spans))

if __name__ == "__main__":
    main()
```

Στην run_intro() πρώτα κατεβάζουμε το “sx-stackoverflow” dataset και το κάνουμε extract αν δεν το έχουμε κάνει ήδη μέσω της download_and_extract_dataset(). Έπειτα μέσω της read_df() φτιάχνουμε τα δεδομένα του σε ένα pandas Dataframe με στήλες “SRC”, “DST” και “UNIXTS”. Έχουμε ορίσει την σταθερά N=5000, ο οποίος είναι ένας υψηλός αριθμός λόγω των απαιτήσεων σε μνήμη (όταν δοκιμάσαμε π.χ. με N=3000 έσκαγε ο κώδικας των επόμενων parts από μνήμη ακόμα και σε μηχάνημα με 32GB RAM). Ο κώδικας υπολογίζει το dt και βρίσκει τα όρια t0, t1, …, t{N-1}, tmax που θα χρησιμοποιηθούν για την δημιουργία των γράφων. Αντί για tmax χρησιμοποιούμε ως τελευταίο σημείο το tmax + 1 ώστε να μπορούμε να χρησιμοποιούμε για ένα δίκτυο πάντα τις ανισότητες t{j-1} <= t < tj με γνήσια ανισότητα για το πάνω όριο χωρίς να ξεχωρίζουμε την τελευταία περίπτωση ως t{j-1} <= t <= tj (δηλαδή με μικρότερο ή ίσο για το πάνω όριο του τελευταίου γράφου).

 # run_intro.py

```
def download_and_extract_dataset(dataset_path: Path):
    if not Path.exists(Path(constants.DATASETS_DIR)):
        Path.mkdir(Path(constants.DATASETS_DIR), parents=True, exist_ok=True)

    dataset_gz_path = Path(constants.DATASETS_DIR).joinpath(
        constants.DATASET_GZ_NAME)
    # download dataset if not exists
    if not Path.exists(dataset_gz_path):
        logging.info(f"could not find {dataset_gz_path}, downloading it")
        request.urlretrieve(constants.DATASET_URL, Path(
            constants.DATASETS_DIR).joinpath(constants.DATASET_GZ_NAME))
```

```
    else:
        logging.info(f"{dataset_gz_path} is already downloaded")

    # extract dataset if not extracted
    if not Path.exists(dataset_path):
        logging.info(f"could not find {dataset_path}, extracting dataset")
        with gzip.open(dataset_gz_path, "rb") as infile:
            with open(dataset_path, "wb") as outfile:
                shutil.copyfileobj(infile, outfile)
    else:
        logging.info(f"{dataset_path}, is already extracted")


def read_df(dataset_path: Path) -> pd.DataFrame:
    logging.info(f"reading {dataset_path} as DataFrame")
    sx_df = pd.read_table(dataset_path, delim_whitespace=True,
                          index_col=None, header=None, names=constants.DFCOLS)
    logging.debug(sx_df.describe())
    logging.debug(sx_df.head())
    logging.debug("%s", sx_df.dtypes)
    return sx_df


def run_intro() -> RunIntroOutput:
    logging.debug("start intro")
    output_dir_path = Path(constants.OUTPUT_DIR)
    if not Path.exists(output_dir_path):
        Path.mkdir(output_dir_path, parents=True, exist_ok=True)
    cache_dir_path = Path(constants.CACHE_DIR)
    if not Path.exists(cache_dir_path):
        Path.mkdir(cache_dir_path, parents=True, exist_ok=True)

    cache_run_intro_output_path = cache_dir_path.joinpath(
        "cache_run_intro_output")

    cache_run_intro_output: RunIntroOutput | None = utils.load_from_cache(
        cache_run_intro_output_path, constants.USE_CACHE_INTRO)
    if cache_run_intro_output is not None:
        return cache_run_intro_output

    dataset_path = Path(constants.DATASETS_DIR).joinpath(
        constants.DATASET_NAME)
    download_and_extract_dataset(dataset_path)

    sx_df = read_df(dataset_path)
    sx_df = sx_df.sort_values(by=constants.DFCOL_UNIXTS, ascending=True)

    srcs: NDArray[int64] = sx_df[constants.DFCOL_SRC].unique()
    dsts: NDArray[int64] = sx_df[constants.DFCOL_DST].unique()

    all_users = np.sort(np.unique(np.concatenate((srcs, dsts), axis=0)))
    logging.debug("all_users=\n%s", all_users)

    unixts: NDArray[int64] = sx_df[constants.DFCOL_UNIXTS].to_numpy()
    t_min: int64 = unixts.min()
    t_max: int64 = unixts.max()
    DT: int64 = t_max - t_min
    # dt: int64 = np.ceil(DT / constants.N)
    dt: float64 = DT / constants.N
```

```
    logging.info(f"t_min={t_min}, t_max={t_max}, DT={DT}, dt={dt}")

    time_spans: list[int64] = [
        int64(np.ceil(t)) for t in np.arange(t_min, t_max, dt, dtype=float64)]
    if time_spans[-1] < t_max:
        time_spans.append(t_max + 1)

    logging.debug("end intro")
    run_intro_output = RunIntroOutput(sx_df, t_min, t_max, DT, dt, time_spans)
    utils.dump_to_cache(cache_run_intro_output_path, run_intro_output)
    return run_intro_output
```

Part 1

Οι υπολογισμοί που ζητούνται στο part1 είναι αρκετά χρονοβόροι και αδύνατο να τους εκτελέσουμε για όλα τα 5000 γραφήματα που θα δημιουργηθούν, για αυτό έχουμε ορίσει την σταθερά N_SHOW_PART1 = 6 με την οποία υπολογίζουμε μόνο 6 γραφήματα και δείχνουμε τα μέτρα για αυτά.

 # run_part1.py

```
    def plot_histogram(data: list, index: int, type: str, plots_dir: Path, block:
bool = False):
    plt.figure(1)
    name = f"Graph{index}_{type}"
    logging.debug(
        f"plotting {name}")
    plt.suptitle(f"{name}")
    # n = 10.0
    # min_d = min(data)
    # max_d = max(data)
    # dn = (max_d - min_d) / n
    # print(f"min{min_d}, max{max_d}, dn{dn}")
    # bins = [abin for abin in np.arange(min_d, max_d, dn)]
    bins = [abin for abin in np.arange(0, 1.0, 0.05)]
    bins.append(1.0)
    plt.hist(data, bins=bins, color="blue")
    plt.xlabel("centrality")
    plt.ylabel("nodes")
    xticks = [axtick for axtick in np.arange(0, 1.0, 0.1)]
    xticks.append(1.0)
    plt.xticks(xticks)
    # counts, bins = np.histogram(data, bins=10)
    # plt.hist(counts[:-1], bins=list(bins), weights=counts)
    figure_file: Path = Path(plots_dir).joinpath(f"{name}.png")
    plt.savefig(figure_file)
    plt.show(block=block)


def create_network(t_low: int64, t_upper: int64, sx_df: DataFrame, index: int) ->
nx.Graph:
    logging.debug(
        f"creating Graph{index} between t_low {t_low} and t_upper {t_upper}")
    sx_in_timespan = sx_df[(sx_df[constants.DFCOL_UNIXTS]
                            >= t_low) & (sx_df[constants.DFCOL_UNIXTS] < t_upper)]

    nodes, _, id_to_index_dict = utils.nodes_from_df(
        sx_in_timespan)
    edges = utils.edges_from_df(sx_in_timespan, id_to_index_dict)
    graph = nx.Graph()
    graph.add_nodes_from(nodes)
    graph.add_edges_from(edges)
    # graph.remove_edges_from(nx.selfloop_edges(graph))

    # graph_dict = utils.graph_dict_from_df(sx_in_timespan)
    # graph = nx.Graph(graph_dict)
    return graph
```

```
def calculate_centralities(graph: nx.Graph, t_low: int64, t_upper: int64, index:
int, part1_output_dir: Path, part1_cache_dir: Path):
    if constants.SHOULD_PLOT_GRAPH:
        logging.debug(
            f"plotting Graph{index} t_low {t_low}, t_upper {t_upper}")
        utils.plot_graph(
            graph, name=f"Graph{index}", plots_dir=part1_output_dir)

    # degree_centrality
    cache_file = part1_cache_dir.joinpath(f"Graph{index}_degree_centrality")
    degree_centrality_list: list | None = utils.load_from_cache(
        cache_file, constants.USE_CACHE_PART1_DATA)
    if degree_centrality_list is None:
        logging.debug(
            f"calculating Graph{index} t_low {t_low}, t_upper {t_upper}
degree_centrality")
        degree_centrality_dict = nx.degree_centrality(graph)
        degree_centrality_list = [
            val for val in degree_centrality_dict.values()]
        utils.dump_to_cache(cache_file, degree_centrality_list)

    plot_histogram(degree_centrality_list, index,
                   "DegreeCentrality", part1_output_dir)

    # closeness_centrality
    cache_file = part1_cache_dir.joinpath(f"Graph{index}_closeness_centrality")
    closeness_centrality_list: list | None = utils.load_from_cache(
        cache_file, constants.USE_CACHE_PART1_DATA)
    if closeness_centrality_list is None:
        logging.debug(
            f"calculating Graph{index} t_low {t_low}, t_upper {t_upper}
closeness_centrality")
        closeness_centrality_dict = nx.closeness_centrality(graph)
        closeness_centrality_list = [
            val for val in closeness_centrality_dict.values()]
        utils.dump_to_cache(cache_file, closeness_centrality_list)

    plot_histogram(closeness_centrality_list, index,
                   "ClosenessCentrality", part1_output_dir)

    # betweenness_centrality
    cache_file = part1_cache_dir.joinpath(
        f"Graph{index}_betweenness_centrality")
    betweenness_centrality_list: list | None = utils.load_from_cache(
        cache_file, constants.USE_CACHE_PART1_DATA)
    if betweenness_centrality_list is None:
        logging.debug(
            f"calculating Graph{index} t_low {t_low}, t_upper {t_upper}
betweenness_centrality")
        betweenness_centrality_dict = nx.betweenness_centrality(graph)
        betweenness_centrality_list = [
            val for val in betweenness_centrality_dict.values()]
        utils.dump_to_cache(cache_file, betweenness_centrality_list)

    plot_histogram(betweenness_centrality_list, index,
                   "BetweenessCentrality", part1_output_dir)

    # eigenvector_centrality
    cache_file = part1_cache_dir.joinpath(
```

```
        f"Graph{index}_eigenvector_centrality")
    eigenvector_centrality_list: list | None = utils.load_from_cache(
        cache_file, constants.USE_CACHE_PART1_DATA)
    if eigenvector_centrality_list is None:
        logging.debug(
            f"calculating Graph{index} t_low {t_low}, t_upper {t_upper}
eigenvector_centrality")
        eigenvector_centrality_dict = nx.eigenvector_centrality(
            graph, tol=0.00001)
        eigenvector_centrality_list = [
            val for val in eigenvector_centrality_dict.values()]
        utils.dump_to_cache(cache_file, eigenvector_centrality_list)

    plot_histogram(eigenvector_centrality_list, index,
                   "EigenvectorCentrality", part1_output_dir)

    # katz_centrality
    cache_file = part1_cache_dir.joinpath(f"Graph{index}_katz_centrality")
    katz_centrality_list: list | None = utils.load_from_cache(
        cache_file, constants.USE_CACHE_PART1_DATA)
    if katz_centrality_list is None:
        logging.debug(
            f"calculating Graph{index} t_low {t_low}, t_upper {t_upper}
katz_centrality")
        katz_centrality_dict = nx.katz_centrality(
            graph, max_iter=100000, tol=1.0)
        katz_centrality_list = [
            val for val in katz_centrality_dict.values()]
        utils.dump_to_cache(cache_file, katz_centrality_list)

    plot_histogram(katz_centrality_list, index,
                   "KatzCentrality", part1_output_dir)


def plot_nodes_or_edges(plots_dir: Path, indexes: list[int], all_nodes_or_edges:
list[int], name: str, block: bool = False):
    plt.clf()
    logging.debug(f"plotting {name}")
    plt.suptitle(f"{name}")
    x = [index for index in range(0, len(all_nodes_or_edges), 1)]
    plt.bar(x, all_nodes_or_edges, 1, color="blue")
    plt.xlabel("graph index")
    plt.ylabel(f"{name}")
    plt.xticks(x, indexes)
    figure_file: Path = Path(plots_dir).joinpath(f"{name}.png")
    plt.savefig(figure_file)
    plt.show(block=block)


def run_part1(run_part1_input: RunPart1Input):
    logging.debug("start part1")
    part1_output_dir = Path(constants.OUTPUT_DIR).joinpath("part1")
    if not Path.exists(part1_output_dir):
        Path.mkdir(part1_output_dir, exist_ok=True, parents=True)
    part1_cache_dir = Path(constants.CACHE_DIR).joinpath("part1")
    if not Path.exists(part1_cache_dir):
        Path.mkdir(part1_cache_dir, exist_ok=True, parents=True)

    if constants.USE_CACHE_PART1:
```

```
        return

    time_spans = run_part1_input.time_spans
    all_nodes: list[int] = []
    all_edges: list[int] = []
    indexes: list[int] = []
    # calculate which indexes to show. The last element is the upper limit of the
last graph so it is excluded.
    show_part1_indexes = utils.parts_indexes_from_list(
        time_spans[:-1], constants.N_SHOW_PART1)
    show_part1_indexes_set = set(show_part1_indexes)
    for index, t_low in enumerate(time_spans):
        if index < len(time_spans) - 1:
            if index in show_part1_indexes_set:
                t_upper = time_spans[index+1]
                graph = create_network(t_low, t_upper, run_part1_input.sx_df,
                                       index)
                indexes.append(index)
                all_nodes.append(len(graph.nodes))
                all_edges.append(len(graph.edges))
                calculate_centralities(graph, t_low, t_upper,
                                       index, part1_output_dir, part1_cache_dir)

    print('indexes', indexes)
    print('all_nodes', all_nodes)
    print('all_edges', all_edges)
    plot_nodes_or_edges(part1_output_dir, indexes, all_nodes, "Nodes")
    plot_nodes_or_edges(part1_output_dir, indexes, all_edges, "Edges")

    logging.debug("end part1")
```

ΑΠΟΤΕΛΕΣΜΑΤΑ:

1. **Partition the complete time period T = [tmin, tmax] into a set of non-overlapping time periods {T1 , . . . , TN } by computing the corresponding set of time instances {t0 , . . . , tN } where t0 = tmin and tN = tmax . Mind that N is a user defined parameter.**

Όπως έχουμε πει έχουμε επιλέξει N=5000 και έχουμε υπολογίσει όλα τα tj από το intro.

2. **Choose an appropriate representation for each subgraph G[tj−1 , tj] of the network for each time period Tj where 1 ≤ j ≤ N .**

Για την δημιουργία των δικτύων χρησιμοποιούμε την βιβλιοθήκη **networkx**. Από αυτή χρησιμοποιούμε την κλάση **Graph** όπου δηλώνει έναν μην κατευθυνόμενο γράφο. Προσοχή το networkx αφήνει να υπάρχουν self directed loops ακόμα και στους μη κατευθυνόμενους γράφους. Στην συνάρτηση create_network() που δείχνουμε παραπάνω περιορίζουμε το dataset μεταξύ του διαστήματος [t_low, t_upper) και παίρνουμε τους κόμβους και τις ακμές αυτού του περιορισμένου dataset με τις παρακάτω συναρτήσεις nodes_from_df() και edges_from_df() που έχουμε φτιάξει:

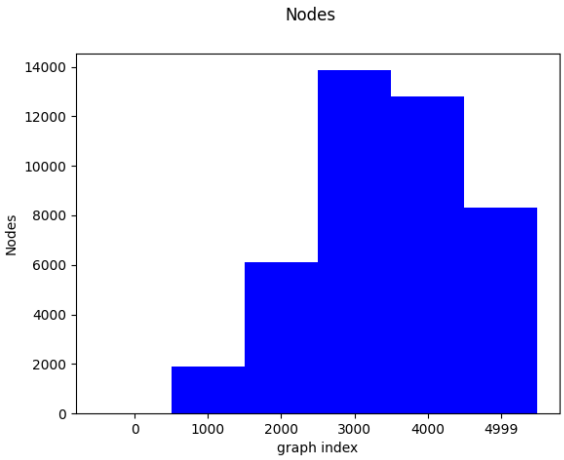
```
def get_id_index_dicts(nodes: NDArray[int64]) -> tuple[dict[int64, int64],
dict[int64, int64]]:
```

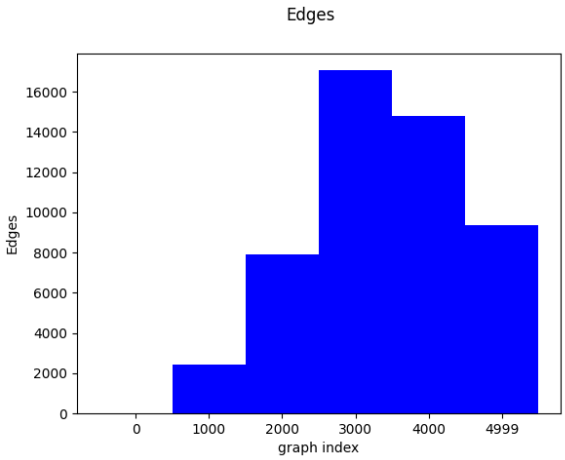
```
    nodes.sort()
    index_to_id_dict: dict[int64, int64] = {}
    id_to_index_dict: dict[int64, int64] = {}
    for index, node in enumerate(nodes):
        index_to_id_dict[int64(index)] = node
        id_to_index_dict[node] = int64(index)

    return index_to_id_dict, id_to_index_dict

def nodes_from_df(sx_df: DataFrame) -> tuple[NDArray[int64], dict[int64, int64],
dict[int64, int64]]:
    srcs: NDArray[int64] = sx_df[constants.DFCOL_SRC].unique()
    dsts: NDArray[int64] = sx_df[constants.DFCOL_DST].unique()

    nodes_ids = np.sort(np.unique(np.concatenate((srcs, dsts), axis=0)))
    index_to_id_dict, id_to_index_dict = get_id_index_dicts(nodes_ids)
    nodes = np.array(range(0, len(nodes_ids), 1))
    return nodes, index_to_id_dict, id_to_index_dict


def edges_from_df(sx_df: DataFrame, id_to_index_dict: dict[int64, int64]) ->
list[tuple[int64, int64]]:
    edges_set: set[tuple[int64, int64]] = set()
    for _, row in sx_df.iterrows():
        source_id = row[constants.DFCOL_SRC]
        target_id = row[constants.DFCOL_DST]
        source = id_to_index_dict[source_id]
        target = id_to_index_dict[target_id]
        edges_set.add((source, target))
    edges = list(edges_set)
    return edges
```

Σημείωση για ευκολία των υπολογισμών χρησιμοποιούμε ως κόμβους τον αύξοντα αριθμό (index) και όχι το userId όπως υπάρχει στο dataset. Μέσω των λεξικών που φτιάχνει η συνάρτηση get_id_index_dicts() μπορούμε ανά πάσα στιγμή να πάρουμε το πραγματικό userId ενός κόμβου.

3. **Provide a graph depicting the time evolution of the quantities |V [tj−1, tj]| and |E[tj−1 , tj]| for each time period Tj where 1 ≤ j ≤ N .**

Δείχνουμε τα μεγέθη σε ιστογράμματα μόνο για τα 6 διαστήματα που κάνουμε τους υπολογισμούς.





4. **For each subgraph G[tj−1 , tj] compute and graphically represent the probability density functions (i.e. histograms of relative frequencies) for the following centrality measures:**

   (a) Degree Centrality

   (b) Closeness Centrality

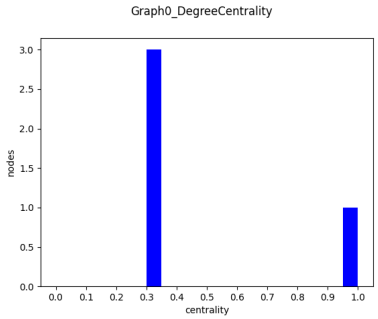
   (c) Betweenness Centrality

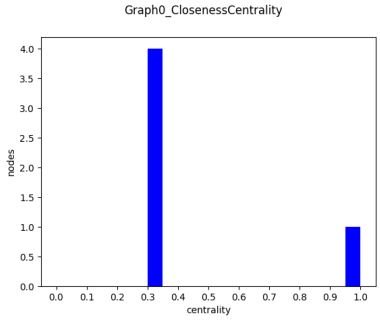
   (d) Eigenvector Centrality

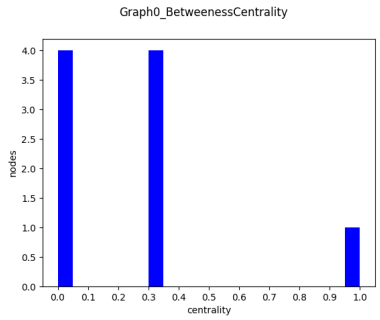
   (e) Katz Centrality

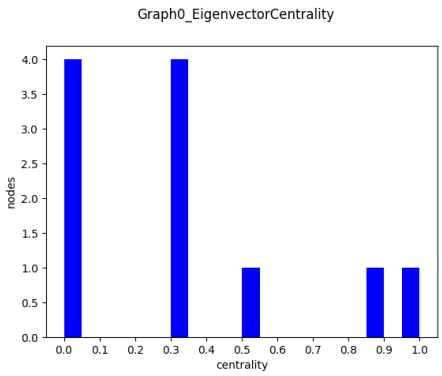
Τα μέτρα κεντρικότητας υπολογίζονται από την συνάρτηση calculate_centralities() για τους 6 γράφους που έχουμε επιλέξει να δείξουμε. Τα γραφήματά τους αποθυκεύονται στο directory “output/part1”. Σε αυτή την αναφορά δείχνουμε μόνο για τον 1ο και τον τελευταίο για λόγους συντομίας:

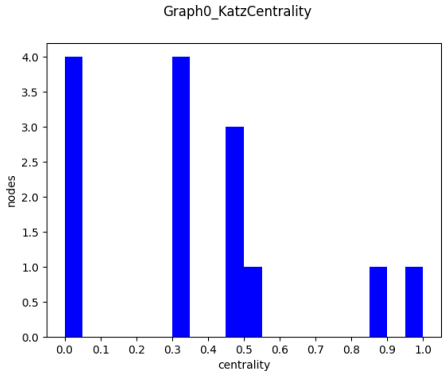
Γράφος0:



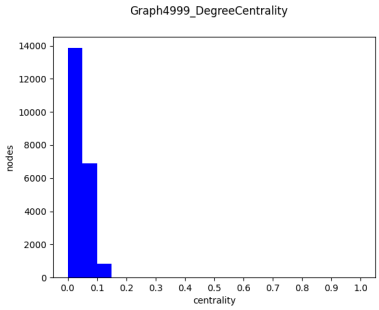


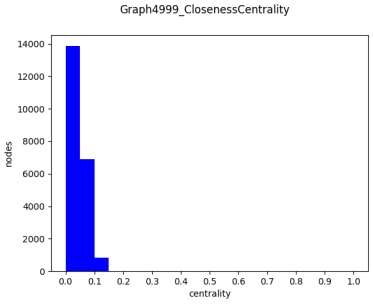


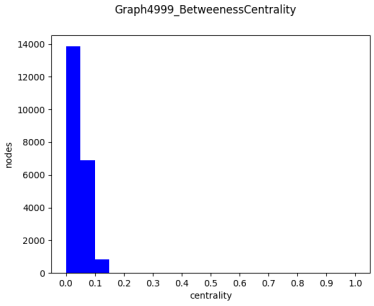

Graph0_EigenvectorCentrality
nodes
centrality

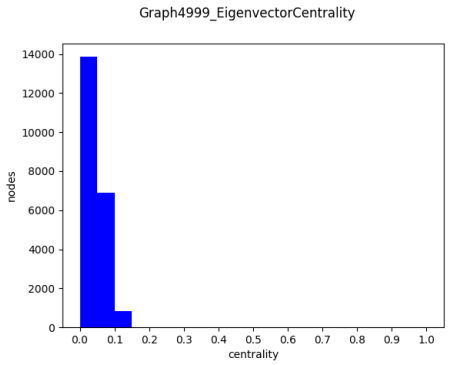

Graph0_KatzCentrality
nodes
centrality

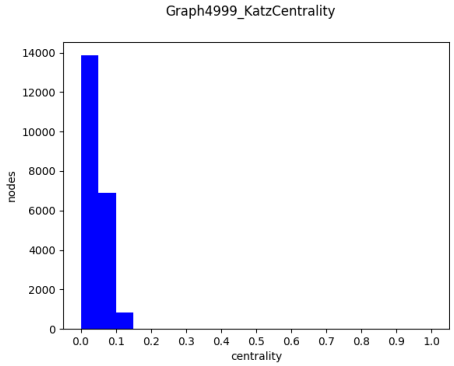
Γράφος 4999:


Graph4999_DegreeCentrality
nodes
centrality


Graph4999_ClosenessCentrality
nodes
centrality


Graph4999_BetweenessCentrality
nodes
centrality


Graph4999_EigenvectorCentrality
nodes
centrality


Graph4999_KatzCentrality
nodes
centrality

Part 2

Οι υπολογισμοί που ζητούνται στο part2 είναι αρκετά χρονοβόροι και αδύνατο να τους εκτελέσουμε για όλα τα 5000 γραφήματα που θα δημιουργηθούν, για αυτό έχουμε ορίσει την σταθερά N_SHOW_PART2 = 6 με την οποία εκτελούμε τους υπολογισμούς μόνο για 6 διαστήματα κόμβων και άρα 12 γραφήματα.

 # run_part2.py

```
def create_networks(t_low: int64, t_mid: int64, t_upper: int64, sx_df: DataFrame,
index: int) -> tuple[nx.Graph, nx.Graph, dict[int64, int64], dict[int64, int64]]:
    logging.debug(
        f"creating Graph{index} [t_low,t_mid]=[{t_low}-{t_mid}] and Graph{index+1}
[t_mid,t_upper]=[{t_mid},{t_upper}]")
    sx_in_timespan = sx_df[(sx_df[constants.DFCOL_UNIXTS]
                            >= t_low) & (sx_df[constants.DFCOL_UNIXTS] < t_upper)]

    sx_in_tlow = sx_in_timespan[(sx_in_timespan[constants.DFCOL_UNIXTS]
                                 >= t_low) &
(sx_in_timespan[constants.DFCOL_UNIXTS] < t_mid)]
    sx_in_tupper = sx_in_timespan[(sx_in_timespan[constants.DFCOL_UNIXTS]
                                   >= t_mid) &
(sx_in_timespan[constants.DFCOL_UNIXTS] < t_upper)]

    nodes, index_to_id_dict, id_to_index_dict = utils.nodes_from_df(
        sx_in_timespan)
    # graph tlow
    edges_tlow = utils.edges_from_df(sx_in_tlow, id_to_index_dict)
    graph_tlow = nx.Graph()
    graph_tlow.add_nodes_from(nodes)
    graph_tlow.add_edges_from(edges_tlow)
    # graph tupper
    edges_tupper = utils.edges_from_df(sx_in_tupper, id_to_index_dict)
    graph_tupper = nx.Graph()
    graph_tupper.add_nodes_from(nodes)
    graph_tupper.add_edges_from(edges_tupper)
    assert np.array_equal(nodes, np.array(graph_tlow.nodes))
    assert np.array_equal(nodes, np.array(graph_tupper.nodes))
    return graph_tlow, graph_tupper, index_to_id_dict, id_to_index_dict


def plot_nodes(plots_dir: Path, indexes: list[int], all_nodes_or_edges: list[int],
name: str, block: bool = False):
    plt.clf()
    logging.debug(f"plotting {name}")
    plt.suptitle(f"{name}")
    x = [index for index in range(0, len(all_nodes_or_edges), 1)]
    plt.bar(x, all_nodes_or_edges, 1, color="blue", edgecolor="yellow")
    plt.xlabel("graph index")
    plt.ylabel(f"{name}")
    indexes_labels = list(map(lambda index: f"T{index}-T{index+1}", indexes))
    plt.xticks(x, indexes_labels, fontsize=8)
    figure_file: Path = Path(plots_dir).joinpath(f"{name}.png")
    plt.savefig(figure_file)
    plt.show(block=block)
```

```
def plot_edges(plots_dir: Path, indexes: list[int], all_edges_tlow: list[int],
all_edges_tupper: list[int], name: str, block: bool = False):
    plt.clf()
    logging.debug(f"plotting {name}")
    plt.suptitle(f"{name}")
    all_edges = []
    for i, _ in enumerate(all_edges_tlow):
        all_edges.append(all_edges_tlow[i])
        all_edges.append(all_edges_tupper[i])
    x = [index for index in range(0, len(all_edges), 1)]
    plt.bar(x, all_edges, 1, color="blue", edgecolor="yellow")
    plt.xlabel("graph index")
    plt.ylabel(f"{name}")
    all_indexes = []
    for index in indexes:
        all_indexes.append(index)
        all_indexes.append(index+1)
    indexes_labels = list(map(lambda index: f"T{index}", all_indexes))
    plt.xticks(x, indexes_labels, fontsize=8)
    figure_file: Path = Path(plots_dir).joinpath(f"{name}.png")
    plt.savefig(figure_file)
    plt.show(block=block)


def calculate_sources_targets(nodes_len: int, part2_cache_dir: Path):
    logging.debug("start calculate_nodes_indexes")
    sources_path = part2_cache_dir.joinpath("sources_column")
    targets_path = part2_cache_dir.joinpath("targets_column")
    if Path.exists(sources_path) and Path.exists(targets_path):
        return

    sources, targets = np.triu_indices(nodes_len)
    utils.dump_to_cache(sources_path, sources)
    utils.dump_to_cache(targets_path, targets)


def calculate_adjacency_matrix(graph: nx.Graph, nodes_len: int, path: Path):
    logging.debug("start calculate_adjacency_matrix")
    if Path.exists(path):
        return

    adjacency_matrix: NDArray[float64] = nx.to_numpy_array(
        graph, nodelist=sorted(graph.nodes()))
    if constants.ENABLE_PART2_VALIDATIONS:
        # assertions for logic consistency
        assert (adjacency_matrix == adjacency_matrix.T).all(
        ), "part2->networks_calculations: adjacency_matrix should be symetrical"

    adjacency_column = utils.convert_symetrical_array_to_column(
        adjacency_matrix, nodes_len)
    utils.dump_to_cache(path, adjacency_column)


def distance_dict_to_ndarray(nodes_len: int, distance_dict: dict[int64, dict[int64,
int64]]) -> NDArray[float64]:
    logging.debug("start distance_dict_to_ndarray")
    distance_array: NDArray[float64] = np.zeros(
        shape=(nodes_len, nodes_len), dtype=float64)
    for source, target_dict in distance_dict.items():
```

```
        for target, value in target_dict.items():
            distance_array[source, target] = float64(value)

    return distance_array


def calculate_dgeodesic_array(graph: nx.Graph, nodes_len: int):
    logging.debug("start calculate_dgeodesic_array")

    dgeodesic = nx.shortest_path_length(graph)
    dgeodesic = dict(dgeodesic)
    dgeodesic = distance_dict_to_ndarray(
        nodes_len, dgeodesic)
    return dgeodesic


def calculate_sdg_array(graph: nx.Graph, nodes_len: int, path: Path):
    logging.debug("start calculate_sdg_array")
    if Path.exists(path):
        return

    sdg_array: NDArray[float64] = np.negative(
        calculate_dgeodesic_array(graph, nodes_len))
    if constants.ENABLE_PART2_VALIDATIONS:
        # assertions for logic consistency
        assert (sdg_array == sdg_array.T).all(
        ), "part2->networks_calculations: sdg_array should be symetrical"
    sdg_column = utils.convert_symetrical_array_to_column(sdg_array, nodes_len)
    utils.dump_to_cache(path, sdg_column)


def calculate_scn_array(graph: nx.Graph, nodes: NDArray[int64], nodes_len: int,
path: Path):
    logging.debug("start calculate_scn_array")
    if Path.exists(path):
        return

    scn_column_len = nodes_len * (nodes_len - 1) // 2 + nodes_len
    scn_column: NDArray = np.zeros(scn_column_len)
    counter = 0
    for i in range(0, nodes_len, 1):
        for j in range(i, nodes_len, 1):
            common_neighbors_len = len(list(
                nx.common_neighbors(graph, i, j)))
            scn_column[counter] = common_neighbors_len
            counter += 1
    # combination n choose 2 plus diagonal
    assert len(scn_column) == scn_column_len == counter

    # scn_column = utils.convert_symetrical_array_to_column(scn_array, nodes_len)
    utils.dump_to_cache(path, scn_column)


def calculate_sjc_array(graph: nx.Graph, nodes_len: int, path: Path):
    logging.debug("start calculate_sjc_array")
    if Path.exists(path):
        return

    sjc_array: NDArray[float64] = np.zeros(
```

```
        shape=(nodes_len, nodes_len), dtype=float64)
    jaccard_coefficient = nx.jaccard_coefficient(graph)
    for u, v, p in jaccard_coefficient:
        sjc_array[u, v] = p
        sjc_array[v, u] = p

    if constants.ENABLE_PART2_VALIDATIONS:
        # assertions for logic consistency
        assert (sjc_array == sjc_array.T).all(
        ), "part2->networks_calculations: sjc_array should be symetrical"

    sjc_column = utils.convert_symetrical_array_to_column(sjc_array, nodes_len)
    utils.dump_to_cache(path, sjc_column)


def calculate_sa_array(graph: nx.Graph, nodes_len: int, path: Path):
    logging.debug("start calculate_sa_array")
    if Path.exists(path):
        return

    sa_array: NDArray[float64] = np.zeros(
        shape=(nodes_len, nodes_len), dtype=float64)
    adamic_adar_index = nx.adamic_adar_index(graph)
    for u, v, p in adamic_adar_index:
        sa_array[u, v] = p
        sa_array[v, u] = p

    if constants.ENABLE_PART2_VALIDATIONS:
        # assertions for logic consistency
        assert (sa_array == sa_array.T).all(
        ), "part2->networks_calculations: sa_array should be symetrical"

    sa_column = utils.convert_symetrical_array_to_column(sa_array, nodes_len)
    utils.dump_to_cache(path, sa_column)


def calculate_spa_array(graph: nx.Graph, nodes_len: int, path: Path):
    logging.debug("start calculate_spa_array")
    if Path.exists(path):
        return

    spa_array: NDArray[float64] = np.zeros(
        shape=(nodes_len, nodes_len), dtype=float64)
    adamic_adar_index = nx.preferential_attachment(graph)
    for u, v, p in adamic_adar_index:
        spa_array[u, v] = p
        spa_array[v, u] = p

    if constants.ENABLE_PART2_VALIDATIONS:
        # assertions for logic consistency
        assert (spa_array == spa_array.T).all(
        ), "part2->networks_calculations: spa_array should be symetrical"

    spa_column = utils.convert_symetrical_array_to_column(spa_array, nodes_len)
    utils.dump_to_cache(path, spa_column)


def networks_calculations(graph_tlow: nx.Graph, graph_tupper: nx.Graph,
id_to_index_dict: dict[int64, int64], part2_cache_dir: Path):
```

```
    logging.debug("start part2 networks_calculations")
    nodes: NDArray[int64] = np.array(graph_tlow.nodes)
    nodes.sort()
    nodes_len: int = len(nodes)
    # nodes pairs
    calculate_sources_targets(nodes_len, part2_cache_dir)
    # tlow
    calculate_adjacency_matrix(
        graph_tlow, nodes_len, part2_cache_dir.joinpath("adjacency_column_tlow"))
    calculate_sdg_array(graph_tlow, nodes_len,
                        part2_cache_dir.joinpath("sdg_column_tlow"))
    calculate_scn_array(
        graph_tlow, nodes, nodes_len, part2_cache_dir.joinpath("scn_column_tlow"))
    calculate_sjc_array(
        graph_tlow, nodes_len, part2_cache_dir.joinpath("sjc_column_tlow"))
    calculate_sa_array(graph_tlow, nodes_len,
                       part2_cache_dir.joinpath("sa_column_tlow"))
    calculate_spa_array(
        graph_tlow, nodes_len, part2_cache_dir.joinpath("spa_column_tlow"))
    # tupper
    calculate_adjacency_matrix(
        graph_tupper, nodes_len,
part2_cache_dir.joinpath("adjacency_column_tupper"))
    calculate_sdg_array(graph_tupper, nodes_len,
                        part2_cache_dir.joinpath("sdg_column_tupper"))
    calculate_scn_array(
        graph_tupper, nodes, nodes_len,
part2_cache_dir.joinpath("scn_column_tupper"))
    calculate_sjc_array(
        graph_tupper, nodes_len, part2_cache_dir.joinpath("sjc_column_tupper"))
    calculate_sa_array(
        graph_tupper, nodes_len, part2_cache_dir.joinpath("sa_column_tupper"))
    calculate_spa_array(
        graph_tupper, nodes_len, part2_cache_dir.joinpath("spa_column_tupper"))

    logging.debug("end part2 networks_calculations")


def run_part2(run_part2_input: RunPart2Input) -> RunPart2Output:
    logging.debug("start part2")
    part2_output_dir = Path(constants.OUTPUT_DIR).joinpath("part2")
    if not Path.exists(part2_output_dir):
        Path.mkdir(part2_output_dir, exist_ok=True, parents=True)
    part2_cache_dir = Path(constants.CACHE_DIR).joinpath("part2")
    if not Path.exists(part2_cache_dir):
        Path.mkdir(part2_cache_dir, exist_ok=True, parents=True)

    cache_file = part2_cache_dir.joinpath("run_part2_output.pkl")
    run_part2_output: RunPart2Output | None = utils.load_from_cache(
        cache_file, constants.USE_CACHE_PART2)
    if run_part2_output is not None:
        return run_part2_output

    time_spans = run_part2_input.time_spans
    print('last-previous', time_spans[-1] - time_spans[-2])
    print('previous-preprevious', time_spans[-2] - time_spans[-3])
    all_nodes: list[int] = []
    all_edges_tlow: list[int] = []
    all_edges_tupper: list[int] = []
```

```
    indexes: list[int] = []
    # calculate which indexes to show. The last element is the upper limit of the
last graph so it is excluded.
    show_part2_indexes = utils.parts_indexes_from_list(
        time_spans[:-1], constants.N_SHOW_PART2)
    # last index is previous index in part2
    show_part2_indexes[-1] = show_part2_indexes[-1] - 1
    show_part2_indexes_set = set(show_part2_indexes)
    nc_output: tuple[NDArray[float64], NDArray[float64], NDArray[float64],
NDArray[float64], NDArray[float64], NDArray[float64],
                     NDArray[float64], NDArray[float64], NDArray[float64],
NDArray[float64], NDArray[float64], NDArray[float64]]

    graph_tlow: nx.Graph | None = None
    graph_tupper: nx.Graph | None = None
    for index, t_low in enumerate(time_spans):
        if index < len(time_spans) - 2:
            if index in show_part2_indexes_set:
                t_mid = time_spans[index+1]
                t_upper = time_spans[index+2]
                graph_tlow, graph_tupper, _, id_to_index_dict =
create_networks(t_low, t_mid, t_upper, run_part2_input.sx_df,

index)
                indexes.append(index)
                all_nodes.append(len(graph_tlow.nodes))
                all_edges_tlow.append(len(graph_tlow.edges))
                all_edges_tupper.append(len(graph_tupper.edges))

                if index == show_part2_indexes[-1]:
                    # if index == 0:
                    networks_calculations(
                        graph_tlow, graph_tupper, id_to_index_dict,
part2_cache_dir)

    print('indexes', indexes)
    print('all_nodes', all_nodes)
    print('all_edges', all_edges_tlow)
    plot_nodes(part2_output_dir, indexes, all_nodes, "Nodes")
    plot_edges(part2_output_dir, indexes,
               all_edges_tlow, all_edges_tupper, "Edges")
    # plot_nodes(part2_output_dir, indexes, all_edges_tupper, "Edges Tj+")

    logging.debug("end part2")
    assert graph_tlow is not None and graph_tupper is not None
    run_part2_output = RunPart2Output(graph_tlow, graph_tupper)
    utils.dump_to_cache(cache_file, run_part2_output)
    return run_part2_output
```

ΑΠΟΤΕΛΕΣΜΑΤΑ:

1. **For each pair of successive network instances (G[tj−1 , tj], G[tj , tj+1]), where 1 ≤ j ≤ N−1, compute the following sets**

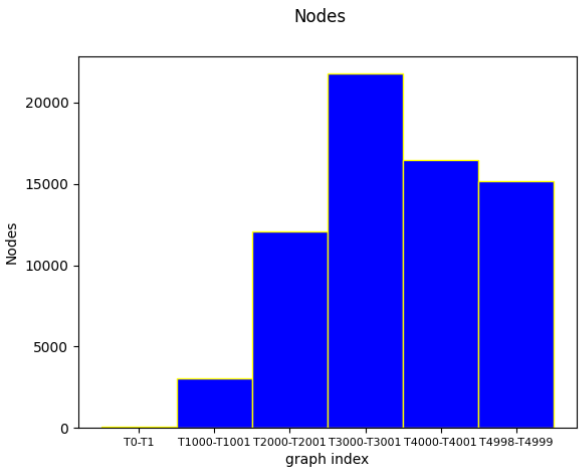
   (a) V [tj−1 , tj+1]

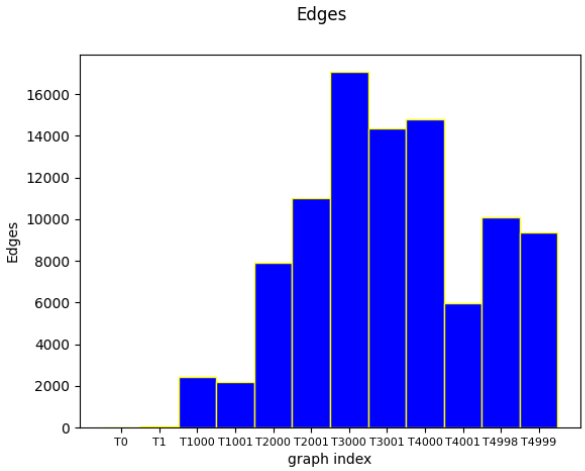
   (b) E∗ [tj−1 , tj]

(c) E∗ [tj , tj+1]

and graphically represent their volumes |V ∗[tj−1 , tj+1]|, |E ∗[tj−1, tj]| and |E ∗[tj , tj+1]| as functions of the coupled time periods (Tj , Tj+1).

Με την συνάρτηση create_networks() φτιάχνουμε 2 γράφους με τους κόμβους που δημιουργούνται στο διάστημα [t{j-1}, t{j+1}) με τις βοηθητικές συναρτήσεις που χρησιμοποιήσαμε και στο part1. Το 1ο γράφημα έχει τις ακμές που δημιουργούνται στο διάστημα [t{j-1}, tj) και το 2ο τις ακμές που δημιουργούνται στο διάστημα [tj, t{j+1}). Δείχνουμε τους κόμβους μόνο για τα 6 διαστήματα που έχουμε ορίσει στην σταθερά N_SHOW_PART2 και τις ακμές για τους 12 γράφους που προκύπτουν από αυτά τα διαστήματα:





2. **For each pair of nodes (u, v) ∈ V ∗ [tj−1 , tj+1] and for every set of common vertices V ∗ [tj−1 , tj+1], where 1 ≤ ≤ N − 1, compute the following imilarity matrices:**

   (a) SGD : SGD (u, v) = −dgeodesic (u, v) [Graph Distance]

   (b) SCN : SCN (u, v) = |Γ(u) ∩ Γ(v)| [Common Neighbors]2

   (c) SJC : SJC (u, v) = |Γ(u)∩Γ(v)| / |Γ(u)∪Γ(v)| [Jaccard's Coefficient]

   (d) SA : SA(u, v) = Σ_z(1 / log(|Γ(z)|)) [Adamic / Adar]

   (e) SPA : S{PA}(u, v) = |Γ(u)| ∗ |Γ(c)| [Preferential Attachment]

Με την συνάρτηση networks_calculations() υπολογίζουμε τα ζητούμενα μεγέθη. Επειδή τα μεγέθη είναι συμμετρικά στους μη κατευθυνόμενους γράφους που έχουμε, κρατάμε μόνο τον άνω τριγωνικό πίνακα (μαζί με την διαγώνιο) σε μορφή στήλης. Δηλαδή αν π.χ. έχουμε n=3 κόμβους, τότε αφού υπολογίσουμε τα μεγέθη σε έναν πίνακα A 3x3 στο τέλος κρατάμε μια στήλη B με αριθμό στοιχείων m=συνδυασμοί n ανά 2 + n (λόγω διαγωνίου) = 3 * (3-2) / 2 + 3 = 6. Σε αυτή την στήλη θα έχουμε

B[0] = A[0][0]

B[1] = A[0][1]

B[2] = A[0][2]

B[3] = A[1][1]

B[4] = A[1][2]

B[5] = A[2][2]

καθώς αν π.χ. έχουμε υπολογίσει ένα μέγεθος από τον κόμβο 1 στον κόμβο 2, τότε δεν μας ενδιαφέρει να κρατήσουμε το μέγεθος από τον κόμβο 2 στον κόμβο 1 λόγω συμμετρίας. Επιπλέον για εξοικονόμηση μνήμης δεν κρατάμε αυτές τις υπολογισμένες στήλες στην μνήμη αλλά τις αποθηκεύουμε σε ένα pickle αρχείο προτού υπολογίσουμε το επόμενο μέγεθος.

Υπολογίζουμε επίσης τον adjacency matrix και κρατάμε με την ίδια λογική όπως πριν μια μόνο στήλη.

Part 3

Στο 3ο μέρος δημιουργούμε ένα νευρωνικό δίκτυο μόνο για τα τελευταία γραφήματα με κόμβους στο [t4998, t5000] και ακμές στο [t4998, t4999) για το 1ο και [t4999, t5000] για το 2ο. Ως features κρατάμε τις στήλες με τις μετρικές που βρήκαμε στο part2 σε έναν πίνακα της μορφής n_κόμβων x 5 και ως labels την στήλη που προέκυψε από τον adjacency matrix. Για train set χρησιμοποιούμε τα μεγέθη που προέκυψαν από τον 1ο γράφο, ενώ για test set τα μεγέθη που προέκυψαν από τον 2ο γράφο. Για λόγους εξοικονόμησης μνήμης δεν χρησιμοποιούμε κάποιο Dataframe αλλά απλούς πίνακες numpy καθαρίζοντας τους από την μνήμη όταν δεν χρειάζονται.

 # run_part3.py

```
def plot_curves(epochs, hist, metrics_names, part3_output_dir: Path):
    """Plot a curve of one or more classification metrics vs. epoch."""
    # metrics should be one of the names shown in:
    #
https://www.tensorflow.org/tutorials/structured_data/imbalanced_data#define_the_mod
el_and_metrics
    plt.figure()
    plt.xlabel("Epoch")
    plt.ylabel("Value")
    for m in metrics_names:
        x = hist[m]
        plt.plot(epochs[1:], x[1:], label=m)
    plt.legend()
    plt.savefig(part3_output_dir.joinpath("train_metrics"))
    plt.show(block=False)
    logging.debug("Defined the plot_curve function.")


def create_model(metrics: list, learning_rate: float) -> tf.keras.Sequential:
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(5,)),
        tf.keras.layers.Dense(32, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1, activation=tf.keras.activations.sigmoid),
        # tf.keras.layers.Dense(2)
    ])
    #   loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                  #
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
                  metrics=metrics)
    return model


def create_train_sets() -> tuple[NDArray, NDArray]:
    logging.debug("start create_train_sets")
    part2_cache_dir = Path(constants.CACHE_DIR).joinpath("part2")
    # sources_column = utils.load_from_cache(
    #     part2_cache_dir.joinpath("sources_column"), True)
    # targets_column = utils.load_from_cache(
    #     part2_cache_dir.joinpath("targets_column"), True)
    sdg_column_tlow: NDArray | None = utils.load_from_cache(
```

```
        part2_cache_dir.joinpath("sdg_column_tlow"), True)
    scn_column_tlow: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("scn_column_tlow"), True)
    sjc_column_tlow: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("sjc_column_tlow"), True)
    sa_column_tlow: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("sa_column_tlow"), True)
    spa_column_tlow: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("spa_column_tlow"), True)
    assert sdg_column_tlow is not None
    assert scn_column_tlow is not None
    assert sjc_column_tlow is not None
    assert sa_column_tlow is not None
    assert spa_column_tlow is not None

    train_features = np.column_stack(
        (sdg_column_tlow, scn_column_tlow, sjc_column_tlow, sa_column_tlow,
spa_column_tlow))
    train_features_normalized: NDArray = preprocessing.normalize(train_features,
axis=0)
    # training_orig_df = pd.DataFrame(train_features, columns=[
    #     constants.NNDFCOL_SRC, constants.NNDFCOL_DST, constants.NNDFCOL_SDG,
constants.NNDFCOL_SCN, constants.NNDFCOL_SJC, constants.NNDFCOL_SA,
constants.NNDFCOL_SPA, constants.NNDFCOL_ADJACENCY])
    # logging.debug("write to_csv")
    # training_orig_df.head(100).to_csv(part3_output_dir.joinpath(
    #     "training_orig_df"), index=False, header=True)

    train_labels = utils.load_from_cache(
        part2_cache_dir.joinpath("adjacency_column_tlow"), True)
    assert train_labels is not None
    logging.debug("end create_train_sets")
    return train_features_normalized, train_labels



def train_model(part3_cache_dir: Path, part3_output_dir: Path, mymodel:
tf.keras.Sequential, metrics_names: list[str], epochs: int, batch_size: int):
    train_features, train_labels = create_train_sets()
    history = mymodel.fit(train_features, train_labels,
                          epochs=epochs, batch_size=batch_size)

    # The list of epochs is stored separately from the rest of history.
    epochs = history.epoch

    # To track the progression of training, gather a snapshot
    # of the model's mean squared error at each epoch.
    hist = pd.DataFrame(history.history)
    plot_curves(epochs, hist, metrics_names, part3_output_dir)


def create_test_sets() -> tuple[NDArray, NDArray]:
    logging.debug("start create_test_sets")
    part2_cache_dir = Path(constants.CACHE_DIR).joinpath("part2")
    sdg_column_tupper: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("sdg_column_tupper"), True)
    scn_column_tupper: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("scn_column_tupper"), True)
    sjc_column_tupper: NDArray | None = utils.load_from_cache(
```

```
        part2_cache_dir.joinpath("sjc_column_tupper"), True)
    sa_column_tupper: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("sa_column_tupper"), True)
    spa_column_tupper: NDArray | None = utils.load_from_cache(
        part2_cache_dir.joinpath("spa_column_tupper"), True)
    assert sdg_column_tupper is not None
    assert scn_column_tupper is not None
    assert sjc_column_tupper is not None
    assert sa_column_tupper is not None
    assert spa_column_tupper is not None

    test_features = np.column_stack(
        (sdg_column_tupper, scn_column_tupper, sjc_column_tupper, sa_column_tupper,
spa_column_tupper))
    test_features_normalized: NDArray = preprocessing.normalize(test_features,
axis=0)
    test_labels = utils.load_from_cache(
        part2_cache_dir.joinpath("adjacency_column_tupper"), True)
    assert test_labels is not None
    logging.debug("end create_test_sets")
    return test_features_normalized, test_labels

def evaluate_model(mymodel: tf.keras.Sequential, batch_size: int, test_features:
NDArray, test_labels: NDArray):
    evaluation = mymodel.evaluate(test_features, test_labels,
batch_size=batch_size)
    logging.info('evaluation')
    print(evaluation)
    return evaluation


def predict_model(mymodel: tf.keras.Sequential, batch_size: int, test_features:
NDArray, test_labels: NDArray):
    logging.debug("start predict_model")
    # probability_model = tf.keras.Sequential([mymodel, tf.keras.layers.Softmax()])
    predictions = mymodel.predict(test_features, batch_size=batch_size)
    predictions[predictions > 0.5] = 1
    predictions[predictions <= 0.5] = 0
    non_zero_indices = np.where(predictions > 0)[0]
    logging.info('test_features[non_zero_indices]')
    logging.info(test_features[non_zero_indices])
    logging.info('predictions[non_zero_indices]')
    logging.info(predictions[non_zero_indices])
    logging.debug("end predict_model")
    return predictions


def run_part3(run_part3_input: RunPart3Input) -> None:
    logging.debug("start part3")
    part3_output_dir = Path(constants.OUTPUT_DIR).joinpath("part3")
    if not Path.exists(part3_output_dir):
        Path.mkdir(part3_output_dir, exist_ok=True, parents=True)
    part3_cache_dir = Path(constants.CACHE_DIR).joinpath("part3")
    if not Path.exists(part3_cache_dir):
        Path.mkdir(part3_cache_dir, exist_ok=True, parents=True)


    learning_rate=0.001
    epochs = 10
```

```
    batch_size = 10000
    metrics = [tf.keras.metrics.Accuracy(name='accuracy'),
tf.keras.metrics.Precision(
        name='precision'), tf.keras.metrics.Recall(name='recall')]
    metrics_names: list[str] = ['loss']
    metrics_names.extend(list(map(lambda metric: metric.name, metrics)))

    mymodel = create_model(metrics, learning_rate)
    train_model(part3_cache_dir, part3_output_dir, mymodel, metrics_names, epochs,
batch_size)
    test_features, test_labels = create_test_sets()
    evaluation = evaluate_model(mymodel, batch_size, test_features, test_labels)
    predictions = predict_model(mymodel, batch_size, test_features, test_labels)
    logging.debug("end part3")
```

Το μοντέλο μας δημιουργείται με την συνάρτηση create_model() που δείχνουμε παραπάνω. Επειδή στην ουσία πρόκειται για ένα binary classification πρόβλημα, στο output layer του νευρωνικού μας δικτύου χρησιμοποιούμε 1 νευρώνα με συνάρτηση sigmoid, ώστε να παίρνουμε απευθείας την πιθανότητα, αντί να χρησιμοποιούμε 2 νευρώνες με onehot-vector κωδικοποίηση και χρήση ενός softmax layer.

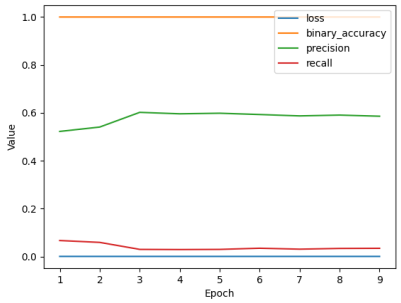
Η συνάρτηση train_model() εκπαιδεύει το μοντέλο μας με τα train sets.

Η συνάρτηση evaluate_model() τεστάρει το μοντέλο μας έναντι των test sets.

Τέλος χρησιμοποιώντας την predict_model() πάνω στα test set παίρνουμε τις πιθανότητες και τις μετατρέπουμε σε 0 ή 1. Έτσι αν π.χ. είχαμε σύνολο κόμβων n=3, τότε στην πέμπτη γραμμή predictions [4] έχουμε την πρόβλεψη 0 ή 1, δηλαδή αν ο κόμβος 1 θα επικοινωνήσει με τον κόμβο 2 στο μελλοντικό διάστημα (έχουμε εξηγήσει προηγουμένως πως προκύπτουν οι κόμβοι από την μετατροπή των μεγεθών σε στήλες στο part2). Θυμίζουμε οι αριθμοί των κόμβων είναι το index τους στους γράφους που είχαμε φτιάξει, αλλά αν χρειαζόμαστε το πραγματικό userId που αντιστοιχεί σε αυτό το index στα πλαίσια υλοποίησης μιας πραγματικής εφαρμογής, τότε μπορούμε να το πάρουμε από τα dictionaries που είχαμε δημιουργήσει κατά την δημιουργία των γραφημάτων.

ΑΠΟΤΕΛΕΣΜΑΤΑ:

Δείχνουμε τις τιμές των μετρικών “loss”, “binary_accuracy”, “precision”, “recall” ανά epoch κατά την εκπαίδευση του νευρωνικού δικτύου:



Δείχνουμε τα αποτελέσματα από την evaluate του μοντέλου μας με τα test sets:

```
evaluation
['loss', 'binary_accuracy', 'precision', 'recall']
[0.0007022073259577155, 0.999958872795105, 0.5484693646430969, 0.045817796140909195]
```