Centrality Analysis of Citation Network of Supreme Court of the United States (SCOTUS)

Michael Kim[1] and James Jushchuk[1]

Department of Statistics and Operations Research
University of North Carolina, Chapel Hill
Submitted for Fall 2016 Review

Advisors: Professor Shankar Bhamidi and Ph.D. candidate Iain Carmichael

Abstract

In this report, we continue our network analysis on the citation network of SCOTUS cases with heavy focus on centrality metrics and modeling. We initially discuss the transition from R into Python for our network analysis, notable Python packages, and web-scraping usage in Python. Then, we show our works for the PyData conference, such as on network visualization, data quality issues, and centrality analysis for legal precedence on SCOTUS. Finally, we show our logistic regression models and how they were assessed using rank-score, which was originally devised by Zanin et al.

1 Network Data Storage in Python[2]

First and foremost, Ph.D. candidate Iain Carmichael and I continued our previous work on the analysis of citation network of the Supreme Court of the United States with plans to present at the PyData Conference during September 15th. The conference was focused on Python tutorials and data science research involving Python. Thus, we thought it would be wise to switch from R to Python for our network analysis and use Python's convenient network packages and web-scraping tools.

We first had to figure out a way to store the case metadata amongst all U.S. cases, which would later serve as the vertex attributes within an interested legal citation network. Using open source data from CourtListener, we accessed their readily available API (https://www.courtlistener.com/api/bulk-info/) and stored each case's metadata as JSON files locally on our computers. Then, we compiled a master CSV file containing all U.S. legal case metadata (over one million cases and all 418 courts/jurisdictions covered **[1]**). The JSON files' information on cited cases also allowed us to create a master edgelist file that contains every

[1] Michael Kim and James Jushchuk are co-authors

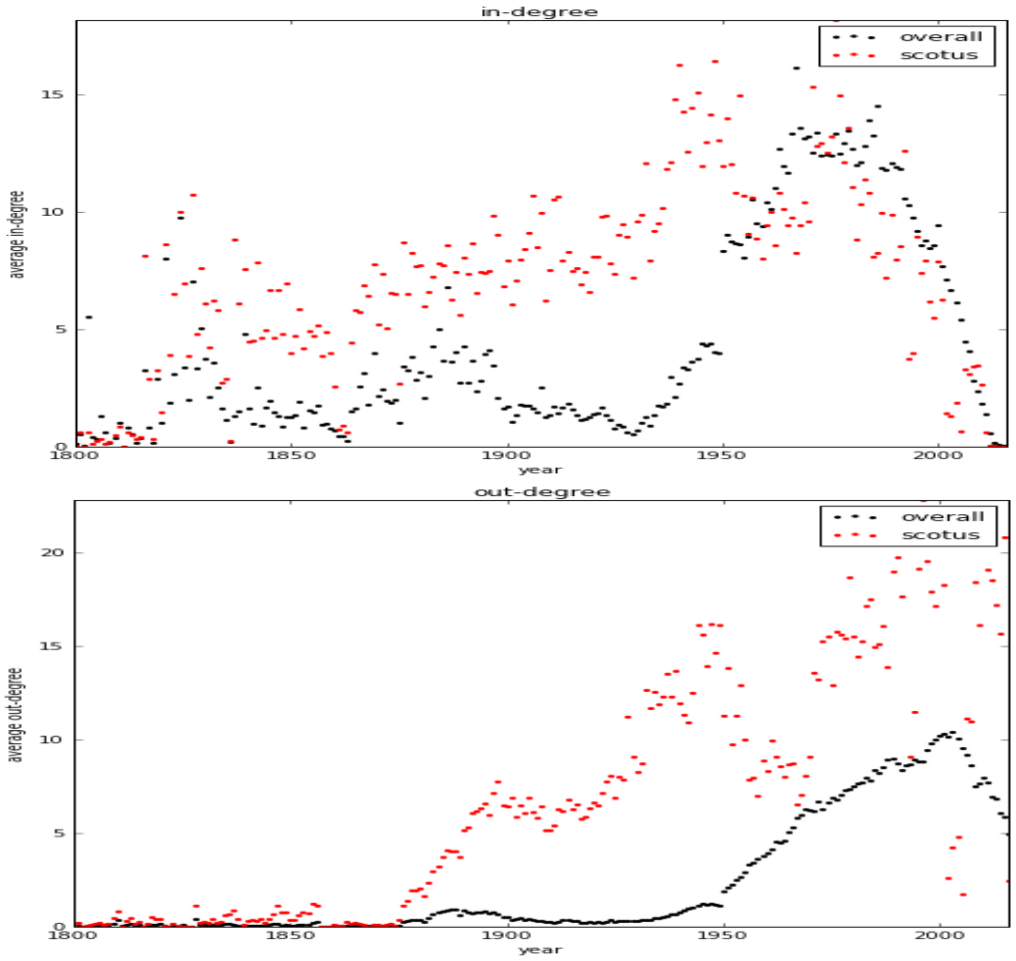
[2] Network data storage work was done mostly by Iain Carmichael, but James and I did contribute to code review and to the understanding of web-scraping in Python

citation relationship between two cases. The Pandas package made it easy to create several other CSV files of case metadata and edgelist that only pertains to a specified jurisdiction, i.e. SCOTUS (33,248 cases and 250,449 edges).

2 Initial Network Exploration

With the Python community's wealth of contribution in the creation of numerous network analysis packages in Python, such as igraph, NetworkX, Graph-Tool, and Graphviz, it was difficult to choose the optimal one. We chose to rely on NetworkX for much of our earlier work due to its well-documented tutorials and guides to using the package and all of its individual functions. For example, every function's purpose were clearly explained and reader friendly to those that may not come from a technical background. Any complex network terms and mathematical equations were properly explained and well-cited. They defined each parameter and their type(s) for every package function in layman fashion, as well as explain the return(s) and return type(s) of the function **[2]**.

Inspired by Fowler's work **[3]**, we decided to reproduce mean indegree (number of citations a case acquires) and outdegree distribution (number of times a case cites other cases) over time for both SCOTUS and the overall U.S. legal citation network.

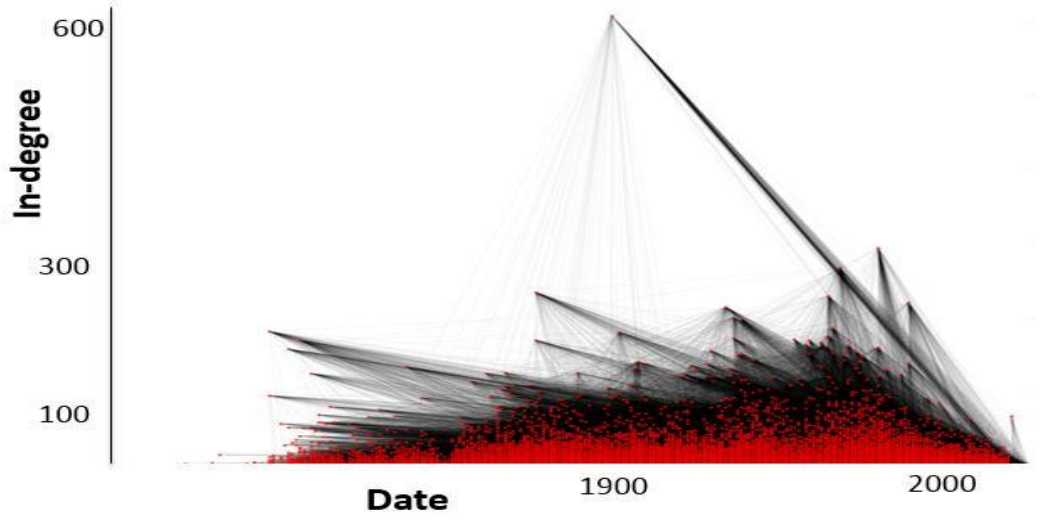


For SCOTUS, both distributions followed a similar trend of overall increase over time with a slight dip near the beginning of the 21st century, which matches Fowler's findings. However, the jump in indegree and outdegree for the overall U.S. legal citation network is much more drastic and happens much earlier compared to the respective degree distributions for SCOTUS, implying that the practice of case citations wasn't as enforced for SCOTUS cases before ~1925.

3 Visualization of SCOTUS Network

When one plans to visualize their network through a default drawing function in a network analysis package, they may encounter long run-times for large networks, such as when using the spring layout in NetworkX (relies on the Fructerman-Reingold force-directed algorithm) for our SCOTUS network with 33,248 nodes and 250,449 edges. This is because the algorithm constantly recalculates the distance every node should be from one another, based on the number of nodes and their degree to effectively visualize the network, which means that the nodes are constantly getting rearranged in their coordinate position as the algorithm processes through each node **[2]**. Thus, even after the visualization processing is complete, most default layouts in most network packages return a clustered ball of nodes and edges that do not convey too much information, especially for larger netowrks.

However, using NetworkX's powerful visualization tools and fixing the nodes' (individual cases) positions by an interested centrality measure and case date allows for quick and meaningful network graph output (in this case, indegree vs. time):



4 Example of Data Quality Problem

We faced several data quality issues in the past couple months on our project of the SCOTUS network, mostly due to the fault of the citation parser of CourtListener. For example, we

identified at least 15 edges that had cases citing forward in time (citing cases in the future) and just recently, we discovered that some of these cases do not contain any opinions or case text.

With the help of our law student collaborator, James Wudel, we noticed that one glaring example of bad data is the U.S. Detroit v. Timber and Lumbar Company with an indegree of approximately 600 (cited ~600 times), which is shown clearly in the plot above. Although we initially believed this to be an extremely important case, as it has around twice as many citations as the second-most cited case, we later discovered that the CourtListener citation parser accidentally picked up a reporter's footnote that refers to this case as a citation. We expressed this problem at the conference **[4]**.

5 Centrality (igraph vs. NetworkX)

Although visualization of networks in scatterplot fashion are easily done in NetworkX due to its well-documented graphing tools, the package faces problems when acquiring the centrality measures for large networks, like this SCOTUS network. Two examples of this are the closeness centrality (takes ~3500 seconds to compute closeness for every SCOTUS case) and betweenness centrality (takes ~19000 seconds to compute betweenness for every SCOTUS case). We believe the reason for the lengthy run times for these two centralities was partly due to the centrality measures having to rely on every other vertex in the network (explained below in section 6.1). This incentivized our research group to switch to igraph, a Python network analysis package which has significantly worse documentation but much faster run times for most of its functions due to most of the algorithms being written under C rather than Python. After this switch, computing closeness and betweenness centralities took mere minutes. We note that they still held significantly longer run times compared to the seconds it takes to compute other centralities. Although we still did most of our visualization work for the PyData Conference through NetworkX, we used mostly igraph for much of our later work, since they do not require fancy visualization. Ultimately, we computed all of our centrality measure for SCOTUS through igraph and saved it as a CSV file for easy, future access.

6 Summary of Centrality Findings Presented at Conference

We focused primarily on how legal precedence could be predicted by centrality measures:

6.1 Closeness Centrality and Betweenness Centrality

As explained by Kolaczyk **[5]** and from our presentation **[4]**, closeness centrality of vertex V is measured by how "close" V is to all other vertices. The denominator below represents the sum of the shortest path distances from V to all other nodes, which means that short distances imply higher centrality.

$$\text{closeness}(\mathbf{V}) = \frac{1}{\sum\limits_{c \in \text{cases}} d(\mathbf{V}, c)}$$

We found out that closeness centrality should be measured on the undirected form of SCOTUS due to its mathematical definition and by viewing its distribution over time. With the help of our

law student collaborator, James Wudel, we found out that closeness centrality tended to favor procedural cases, cases that explain how bodies of law work or operate **[4]**. Betweenness centrality (how "between" V is to all other vertices) was also measured on undirected SCOTUS and favored procedural cases.

6.2 PageRank

On the directed network, we found out that PageRank favored "older cases that precede important cases" **[4]**. More about PageRank will be explained later.

6.3 Hubs/Authorities and Eigenvector Centrality

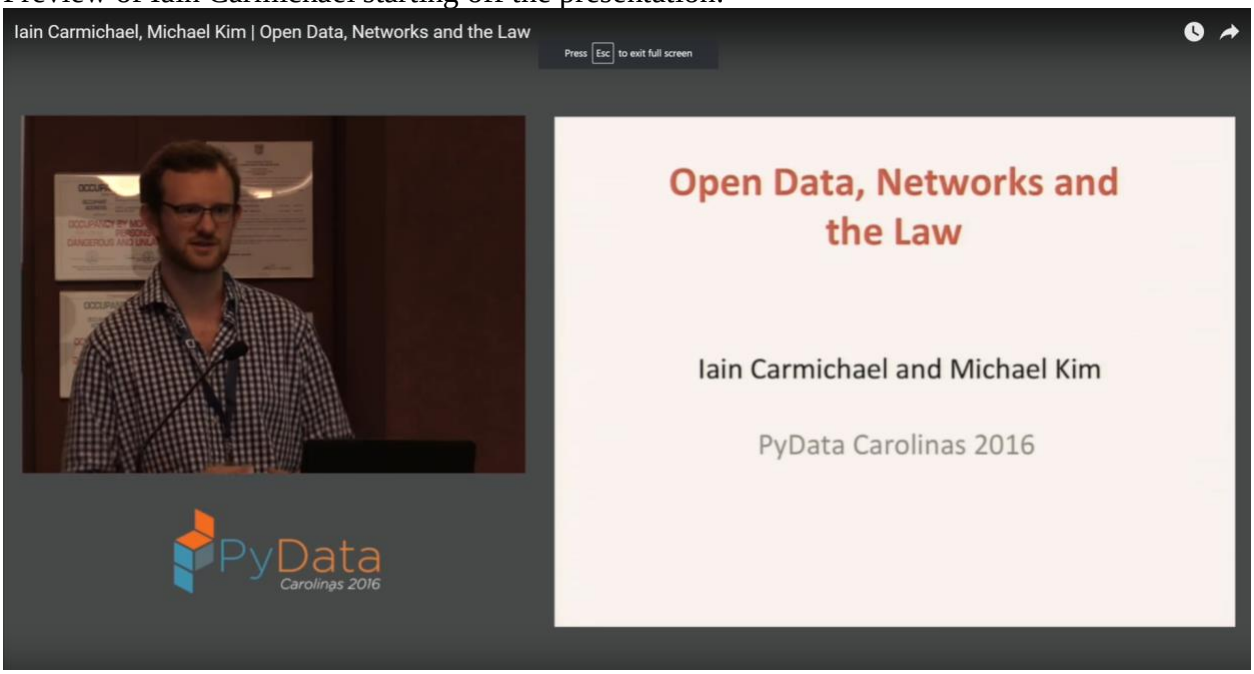
Both centrality measures both concern cases that are cited by other "important" cases, so the definition is recursive in nature. However, as noted by the NetworkX documentation, Hubs/Authorities is to be measured on directed SCOTUS, whereas Eigenvector Centrality is to be measured on undirected SCOTUS **[2]**. Surprisingly, both centrality measures favored cases concerning the First Amendment **[4]**.

6.4 Top Ten cases by Centrality Measures (Case-ID's shown correspond to JSON files)

	In-Degree	**Out-Degree**	**Closeness Centrality**	**Eigenvector Centrality**	**Betweenness Centrality**	**Page Rank**	**Hubs**	**Authorities**
Rank 1	96405	105210	102605	106514	96405	85131	108611	103355
Rank 2	109532	104616	103012	103355	103012	91573	107082	106514
Rank 3	107252	106366	101864	109380	101864	85534	109380	103243
Rank 4	91573	102224	106366	106761	102605	88661	106267	106761
Rank 5	106545	108329	96405	103243	106366	85160	108839	103347
Rank 6	111221	101864	101894	107082	101894	89675	108798	105746
Rank 7	102605	106267	85330	103347	106447	98094	109505	105751
Rank 8	103012	106548	106170	105751	104616	88804	109836	102991
Rank 9	103355	97966	106447	105746	106545	85330	149702	103870
Rank 10	106761	108221	103950	106142	106548	87010	108329	101097

The full presentation can be found here:
https://www.youtube.com/watch?v=AP7_godzwVI&t=937s

Preview of Iain Carmichael starting off the presentation:



7 Motivation for PageRank (and Indegree) for Our Later Models

We found it necessary to use metrics to model and evaluate how influential certain cases are in the large networks we created from the citations of legal cases. Indegree, PageRank, closeness, and betweenness are all metrics that are supported by the igraph coding library, however, the runtimes of these metrics affect how applicable they are to our research. A simple block of code (see below) we ran shows each of these metrics on the same SCOTUS network, and it reveals the disparity in runtimes:

```
IGRAPH DN-- 33248 250465 --
+ attr: court (v), name (v), year (v)
Runtime of PageRank: 0.151999950409 seconds
Runtime of Indegree: 0.00100016593933 seconds
Runtime of Betweenness: 46.896999836 seconds
Runtime of Closeness: 173.968999863 seconds
```

Clearly closeness and betweenness take much longer than the other two metrics to execute, and when we require these metrics to be run repeatedly their runtimes can become impractical. Although indegree appears to be even faster than PageRank, PageRank provides a different interpretation of a case in the network that takes into account more than simply the number of neighbors it has. If indegree is thought of as the number of cases that "vote" (i.e. cite) a particular case, then PageRank can be through of as taking into account that "votes cast by [cases] that are themselves 'important' weigh more heavily and help to make other [cases]

'important'"**[6]**. For now, both indegree and PageRank are valuable metrics and are worth implementing in our research due to their very efficient runtimes.

8 Implementing the PageRank Algorithm

We wanted to implement the PageRank algorithm ourselves, even though a perfectly good implementation already existed in the igraph library because the igraph implementation lacked customization. That implementation had limited options to change the core of the PageRank algorithm, and in our research the ability to tweak the algorithm would be beneficial, i.e. we want to innovate a modified one that takes into account for time decaying patterns in the network, so PageRank doesn't heavily favor older cases all the time in a DAG (directed acyclic graph).

Murphy's *Machine learning: A probabilistic perspective* **[7]** was invaluable for coding our version of the PageRank algorithm, and it outlined all the steps required that we detail in this section. As described in Murphy's text, PageRank can be thought of as the probability a path through the network ends up at a particular node, if the path is determined by following citations of the current case with a small change of randomly jumping to any case in the network. This process of creating paths through the network can be encapsulated by a transition matrix *M*. This matrix *M* is the core design of the PageRank algorithm. *M* is then used to find its first eigenvector. This eigenvector is of size 1x*n* where *n* is the number of cases in a network, and each element in this vector is a PageRank of a case. So the largest challenge of implementing the PageRank algorithm was creating the matrix *M*, which is of the form **[7]**:

$$M_{ij} = \begin{cases} pG_{ij}/c_j + \ \delta & \text{if } c_j \neq 0 \\ 1/n & \text{if } c_j \neq 0 \end{cases}$$

- Where p is the probability at any given case the next case in the path is one of the outward citations (i.e. there is a $1 - p$ probability that the next case will be a random one in the network)
- G_{ij} is 1 if there exists a citation from case *i* to case *j*
- c_j represents the outdegree (number of citations) of case *j*
- n is the number of cases
- δ is of the form $(1 - p)/n$.

To develop this transition matrix, we first had to receive as input an adjacency matrix of a given citation network. This represents *G*, and we also defined n, p, and δ. The next step was to efficiently represent *M* by defining two more variables: a diagonal matrix *D* and a vector z. *D* was created by defining the element d_{jj} as $1/c_j$ if there case *j* had any outward citations, and 0 otherwise. *z* was created by defining each element z_j as δ if the same condition as above was true (case *j* has a least one outward citation) and 0 is case *j* did not have any outward citations. With all these variables defined, we could compactly define *M* as **[7]**:

$$\mathrm{M} = p\mathrm{GD} + 1z^T$$

Lastly, we had to apply the power method to determine the first eigenvector of *M*. This relatively simple process involved us calculating the product of *Mv*, where *v* is an arbitrary vector of length *n*. That product becomes the new *v*, which is once again multiplied with *M*. This process continues until the product *v* has converged. From out tests this would occur around 14 iterations of the process for a small network and up to 70 iterations for a large network such as all SCOTUS cases. This converged vector *v* would then contain the PageRank of each case in the network **[7]**.

Because Murphy did such an excellent job defining the PageRank algorithm in his textbook, the largest challenge we faced was defining all the variables in the proper form. For example, the adjacency matrix we received as input had a value of 1 in the element A_{ij} if case *i* cited case *j*, but when it was defined in Murphy's text, it was the opposite relation. That required us to transpose the matrix *A*. Although our implementation of PageRank was functionally correct (we confirmed this by comparing our PageRank values with igraph's), it had a slower runtime. It would take factors longer than igraph's implementation, and so even though we wanted more customization of PageRank for our research, it ended up being the case the runtime was too slow to justify the benefits of customization. Still, implementing the PageRank algorithm was a valuable lesson in understanding how the metrics we use work, and practicing proper coding guidelines (commenting, proper variable naming) while interpreting explanations of code from textbooks.

Our code for PageRank Implementation can be found at: https://github.com/idc9/law-net/blob/jamesjushchuk/explore/James/pagerank.ipynb

9 Data Frames, Logistic Regression, Rank-Score

After the PyData conference, our group focused on working towards a novel, legal case ranking system based on statistics/machine-learning techniques, such as logistic regression and cross validation. We planned to build numerous models and select the best one through the rank-score system inspired by Zanin, et al. **[8]**.

The procedure in building our logistic regression models on different combinations of centrality metrics is complex, so the steps will be laid out in chronological order:

1. We created SCOTUS subgraphs G_{1900}, G_{1910}, …, G_{2020}, where G_{1900} indicates the SCOTUS subgraph that only includes cases with years up to and including 1900. Therefore, G_{2020} would be the entire SCOTUS network.

```
1900 :  10446 vertices and  25673  edges
1910 :  12463 vertices and  37863  edges
1920 :  14880 vertices and  52273  edges
1930 :  16887 vertices and  67360  edges
1940 :  18585 vertices and  86575  edges
1950 :  20079 vertices and  106643  edges
1960 :  21329 vertices and  118368  edges
1970 :  23642 vertices and  136683  edges
1980 :  25734 vertices and  166571  edges
```

```
1990 :  27848 vertices and  199816  edges
2000 :  29206 vertices and  221711  edges
2010 :  32505 vertices and  238165  edges
2020 :  33248 vertices and  250449  edges
```

2. We created vertex data frames for each of the SCOTUS subgraphs above, which consists of the vertices and their centrality metrics. For now, we created vertex data frames only including indegree and PageRank, but we do intend to later include more centrality metrics along with accounting for time-decay for some of these metrics, especially for indegree and PageRank. Here is the vertex data frame for G_{1950} (V_DF_{1950}).

```
         name    year  indegree  pagerank
0      100000  1922         1  0.000022
1      100001  1922         1  0.000025
2      100002  1922         4  0.000022
3      100003  1922         3  0.000023
4      100004  1922         4  0.000028
5      100005  1922         6  0.000029
...
20073   99994  1922         6  0.000052
20074   99995  1922         1  0.000020
20075   99996  1922         0  0.000020
20076   99997  1922         1  0.000021
20077   99998  1922         6  0.000029
20078   99999  1922         3  0.000031

[20079 rows x 4 columns]
```

3. We created one edgelist data frame, which includes all 250,449 edges in our SCOTUS network and another 250,449 edges that are non-present in our SCOTUS network. In reality, there are millions of edges that aren't present in our network, when we consider every combination of edge amongst all the vertices. However, to save on run-time when compiling this edgelist data frame, we thought an approximation that only contains 250,449 non-present edges is sufficient for our logistic regression method later. There are four + number of interested metrics columns for our edgelist. In our case, the default four columns would be the "edge," "citing_name," "cited_name," and "cited_year." The interested metric columns would be the "cited_indegree" and "cited_pagerank."

Using igraph's convenient function to get the entire present edges in our network, "G.get_edgelist()," acquiring the 250,449 present edges and their respective information, such as "citing_name," "cited_name", and "cited_year" was a straightforward process. As the names imply, "citing_name" is the citing case, "cited_name" is the cited case, and "cited_year" is the cited case's year. To denote that these 250,449 edges were present in our SCOTUS network, we denoted their "edge" status as 1. To access the metrics, "cited_indegree" and "cited_pagerank," we used the citing case's year for each edge and acquired the respective vertex data frame. For example, if the citing case's year in one of the edges was 1948, then the decade of the vertex data frame we would be interested in would be 1950. Therefore, we would access the V_DF_{1950} to get the "cited_indegre" and "cited_pagerank" because we know that vertex dataframe will contain all the cited cases (from the citing case) and their metric information:

```
#determine which vertex_df to retrieve
decade = citing_year + (10 - citing_year%10)
vertex_df = vertex_df_dict[decade]
```

This took approximately 199.434 seconds, according to one of our runs on IPython notebook.

Acquiring the other 250,449 non-present edges wasn't as straightforward as using igraph's built-in function, "get_edgelist()." Instead, we had to take into account of several factors. First, we randomly picked two vertices from our pool of vertices without replacement and created an edge between these two randomly selected vertices. The non-replacement selection of vertices made sure that the edges we select are all unique from each other. This random edge was then checked to see if it was actually a present edge in the graph (our goal is acquiring non-present edges!) and whether the citing vertex's year was greater than or equal to the cited vertex's year (we do not want edges that represent citations that go forward in time). Their "edge" status was labeled as 0 to represent that they were non-present in our SCOTUS network, and all other interested information, such as "citing_name," "cited_name," "cited_year," "cited_indegree," and "cited_pagerank" were all acquired in similar fashion like before. This took approximately 556.497 seconds, according to one of ours runs on IPython notebook.

Here is a snippet of our compiled edgelist:

	edge	citing_name	cited_name	cited_year	cited_indegree	cited_pagerank
0	0	105146	94388	1896	2	0.000021
1	0	111030	99966	1922	20	0.000049
2	0	106778	84925	1809	0	0.000016
3	0	110601	109319	1975	0	0.000012
4	0	104605	87232	1859	1	0.000020
5	0	1087630	101060	1927	0	0.000016

4. Using the Python package sklearn (sci-kit), we ran logistic regression once on our edgelist data frame. A good reference for logistic regression is looking at the stock market example in 4.3 of *An Introduction to Statistical Learning with Applications in R* by James, Witten, Hastie, and Tibshirani **[9]**. As noted by James, et al. logistic regression allows us to calculate the probability that a non-quantitative (qualitative) response variable falling into some category, in contrast to traditional regression that predicts some quantitative value of an interested response variable. In our case, our response variable is whether or not an edge is present between two vertices, denoted as 0 or 1 under the edge column of the edgelist data frame. Although binary, qualitative response variable can be predicted through linear regression, for the purpose of acquiring probability measures that fall between 0 and 1 to use for our rank-scoring implementation later, logistic regression seemed more appropriate for our situation. The logistic function for one predictor variable as introduced by James, et al. is:

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}$$

,where β_0 and β_1 denote the coefficients that best fit the model and are calculated from given X (predictor) and Y (response) training data and using maximum likelihood estimator.

Our models are based off of logistic regression being ran on different training sets. Our Y_training_set (response), which is our response variable of whether or not an edge is present would not change in this process. However, our X_training_set (predictor), changes depending on different combinations of interested centrality metrics. Thus, modifying the X_training_set is responsible for producing different logistic regression models. Then, after performing logistic regression and acquiring the beta coefficients from some X and Y training sets, we can use these coefficients and the logistic function to acquire the probability of an edge that is present in our network on some X_testing_set.

5. Step-by-Step Procedure on How We Selected the Best Model:

1. To compare our models, we first picked 1000 random cases, R_1, R_2, …, R_{1000}.
2. For one of these cases (call this R_j), we acquired the vertex data frame corresponding to case R_j's year.
3. Then we acquired the attachment probabilities with respect to the interested metric(s) for all the vertices in the vertex data frame by using the logistic regression coefficients and the logistic function.
4. We added these probabilities as a separate column to the vertex data frame and sorted the rows of the vertex data frame by these probabilities in decreasing order (vertices with highest metric probabilities are near the top).
5. This allowed us to use the rank-score system devised by Zanin, et al. **[8].**
    a. We acquired R_j's neighbors (cases Rj cites—they would obviously be in the vertex data frame).
    b. Then we ranked these neighbors by their respective indices+1 in the vertex data frames (so we essentially ranked them by their metric probabilities because the vertex data frame is sorted this way)
    c. We gave a score for each neighbor:

$$score_{neighbor} = 1 - \frac{rank}{total\ number\ of\ cases\ that\ come\ before\ R_j}$$

    d. The score for R_j:

$$score_{R_j} = \sum_{all\ neighbors} score_{neighbor}$$

    e. After we acquired the ranks for R_1, …, R_{1000}, we can calculate the score for the centrality metric(s):

$$metric(s)\ score = \sum_{j=1}^{1000} score_{R_j}$$

We acquired this results for the metric(s) scores:

```
Indegree
this took 9.77799987793 seconds
Score: 2998.94088151

PageRank
this took 9.22500014305 seconds
Score: 2563.98054996

Indegree and PageRank
this took 10.3090000153 seconds
Score: 2970.85944083
```

A snippet of our code for logistical regression and rank-scoring can be found in Appendix A.

The implementation of the rest of our code can be found at:

1. https://github.com/idc9/law-net/blob/jamesjushchuk/explore/James/module1_vertex_data.ipynb
2. https://github.com/idc9/law-net/blob/jamesjushchuk/explore/James/module2_edge_data.ipynb
3. https://github.com/idc9/law-net/blob/jamesjushchuk/explore/James/module3_fit_logistical_regression.ipynb
4. https://github.com/idc9/law-net/blob/jamesjushchuk/explore/James/module4_get_ranking_metrics.ipynb

10 Future Plans

We intend to improve our algorithm so it may run on bigger networks, such as on the overall U.S. legal citation network. Furthermore, our logistic regression and rank score models relied on decade approximations, where we only looked at the vertex data frames that represented decade jumps in the SCOTUS network. We hope to make our models more precise and rely less on approximations through looking at the vertex data frames that represent possibly single year jumps in the SCOTUS network. With some algorithm improvemnts, we hope to test our implementation for models of more combinations of metrics than just indegree and PageRank, such as those analyzed during the PyData conference (closeness, betweenness, eigenvector, hubs/authorities) as well as test on metrics that take into account of time-decay. For example, we may not want to account citations that are greater than 20 years apart because we may want to denote that cases lose too much relevance after some significant amount of time (i.e. the cited case may acquire an indegree of 1 from the citing case if and only if $citing_{case_{year}} - cited_{case_{year}} \leq some\ threshold\ (i.e. 10\ years)$). We also hope to implement centrality measures that aren't built into igraph and test these as models on our logistic-regression/rank-score algorithm, such as alpha centrality and dynamic centrality **[10]**.

11 Appendix A

```
def compute_ranking_metrics(G, logistic_regression_object, columns_to_use,
path_to_vertex_metrics_folder, year_interval, R):
    '''
    Computes the rank score metric for a given logistic regression object.

    Parameters
    ------------
    G: network (so we can get each cases' ancestor network)

    logistic_regression_object: a logistic regression object (i.e. the output
of fit_logistic_regression)

    columns_to_use: list of column names of edge metrics data frame that we
should use to fit logistic regression

    path_to_vertex_metrics_folder: we will need these for prediciton

    year_interval: the year interval between each vertex metric .csv file

    R: how many cases to compute ranking metrics for

    Output
    -------
    The average ranking score over all R cases we tested
    '''

    #select cases for sample
    vertices = set(G.vs)
    cases_to_test = random.sample(vertices, R)

    cases_to_test_rank_scores = []

    #load all the vertex metric dataframes into a dict so they only have to
be read in once
    all_vertex_metrics_df = glob.glob(path_to_vertex_metrics_folder +
"/vertex_metrics*.csv")
    vertex_metric_dict = {}
    for vertex_metrc_df in all_vertex_metrics_df:
        #add df to dict with filepath as key
        vertex_metric_dict[vertex_metrc_df] = pd.read_csv(vertex_metrc_df,
index_col=0)
```

```
    #calculate each case's score
    for case in cases_to_test:

        #determine which vertex_df to retrieve
        year = case['year'] + (year_interval - case['year']%year_interval)

        #look-up that dataframe from given path
        vertex_df = vertex_metric_dict[path_to_vertex_metrics_folder +
'\\vertex_metrics_' + str(year) + '.csv']

        #create df that the logistical regression object will evaluate
        x_test_df = vertex_df[columns_to_use]
        attachment_p = get_attachment_probabilty(logistic_regression_object,
x_test_df)

        # add the attachment probabilities as column
        vertex_df['attachment_p'] = attachment_p
        # sort by attachment probabilities
        vertex_df = vertex_df.sort_values('attachment_p', ascending=False,
kind='mergesort')#.reset_index(drop=True)

        # get neighbors
        neighbors = G.neighbors(case.index, mode='OUT')

        # rank and score neighbors using dataframe indices
        scores = [] # list of scores for each vertex
        for i in neighbors:
            rank = vertex_df.index.get_loc(G.vs[i]['name']) + 1
            score = 1-rank/len(attachment_p)
            scores.append(score)

        case_rank_score = sum(scores) # sum up the scores for each case

        #add score to list of all cases' scores
        cases_to_test_rank_scores.append(case_rank_score)

    return np.mean(cases_to_test_rank_scores)
```

In [57]:

```
def get_attachment_probabilty(logistic_regression_object, x_test_df):
    '''
    Evaluates our logistic regression model for a given dataset.

    Parameters
    ------------
```

```
    logistic_regression_object: a logistic regression object (i.e. the output
of fit_logistic_regression)

    x_test_df: columns of vertex_df used in evaluating the logistical
regression

    Output
    ------
    returns a list of attachment probabilities for the dataset
    '''

    # get attachment probabilities on testing set
    prob = logistic_regression_object.predict_proba(x_test_df)

    # predicted probabilities for ALL case for edge present (1)
    prob_present = prob[:,1:2]
    # convert to list
    prob_present_list = [i.tolist()[0] for i in prob_present]

    return prob_present_list
```

Testing above defs

In [23]:

```
def fit_logistic_regression(path_to_edge_data_frame, columns_to_use):
    '''
    Fits our logistic regression model. Any data you need for logistic
regression should be in the edge data frame

    Parameters
    ------------
    path_to_edge_data_frame:

    columns_to_use: list of column names of edge metrics data frame that we
should use to fit logistic regression

    Output
    ------
    returns a logistic regression object
    '''
    #set up training data
    df = pd.read_csv(path_to_edge_data_frame, index_col=0)
    y_train = df['edge']
    x_train = df[columns_to_use]

    #calculate logistical regression
```

```
    clf = skl_lm.LogisticRegression(solver='newton-cg')
    clf.fit(x_train, y_train)
    return clf
```

In [24]:

```
#This def is not required, I just used it to make excuted code concise
def load_scotus_graph():
    G = load_citation_network_igraph(data_dir, court_name)
    all_edges = G.get_edgelist() # list of tuples
    bad_edges = []
    for edge in all_edges:
        citing_year = G.vs(edge[0])['year'][0]
        cited_year = G.vs(edge[1])['year'][0]

        if citing_year < cited_year:
            bad_edges.append(edge)

    G.delete_edges(bad_edge
```

12 References

[1] "CourtListener." *CourtListener*. Free Law Project, n.d. Web. 14 Dec. 2016. <https://www.courtlistener.com/api/bulk-info/>.

[2] "NetworkX 1.11 Documentation." NetworkX. N.p., n.d. Web. 14 Dec. 2016. <http://networkx.readthedocs.io/en/networkx-1.11/>.

[3] Fowler, James H., and Sangick Jeon. "The Authority of Supreme Court Precedent." Social Networks 30.1 (2008): 16-30. ScienceDirect. Web. 14 Dec. 2016.

[4] Carmichael, Iain, and Michael Kim. "Open Data, Networks, and the Law." PyData Carolinas 2016, 4 Oct. 2016. Web. 14 Dec. 2016. <https://www.youtube.com/watch?v=AP7_godzwVI&t=937s>.

[5] Kolaczyk, Eric D. *Statistical Analysis of Network Data Methods and Models*. New York: Springer, 2009. Print.

[6] Sullivan, Danny. "What Is Google PageRank? A Guide For Searchers & Webmasters." *Search Engine Land*. N.p., 26 Apr. 2007. Web. 14 Dec. 2016.

[7] Murphy, Kevin P. *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT, 2012. Print.

[8] Zanin, Massimiliano, Pedro Cano, Oscar Celma, and Javier M. Buldú . "Preferential Attachment, Aging And Weights In Recommendation Systems." International Journal of Bifurcation and Chaos 19.02 (2009): 755-63. Web.

[9] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning: With Applications in R. New York: Springer, 2013. Print.

[10] Lerman, Kristina, Rumi Ghosh, and Jeon Hyung Kang. "Centrality Metric for Dynamic Networks." Proceedings of the Eighth Workshop on Mining and Learning with Graphs - MLG '10 (2010): n. pag. Web.