Word Contextualization of Various Clusters and Deep Learning Classification of the SCOTUS Citation Network and Text Data

Michael Kim[1], Scott Garcia[1], James Jushchuk[1], and Ethan Koch[1]

Department of Statistics and Operations Research
University of North Carolina, Chapel Hill
Submitted for Spring 2017 Review

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

Abstract

In this report, we explain the various clustering algorithms (k-means, gaussian mixture models, hierarchical clustering) used on the SCOTUS network data and various forms of SCOTUS NLP data (tf-idf matrix, bag-of-words matrix, singular value decomposition of tf-idf matrix, non-negative matrix factorization of tf-idf matrix). The purpose was to generate "summaries" of these clusters (sets of opinions) by extracting words or opinions with significant values from tf-idf matrix, i.e. words with highest tf-idf values. In other words, this was a simple approach to naively contextualize a set of SCOTUS cases with words or the most "relevant" opinion. The "summaries" are presented in a shiny app (https://scottgarcia.shinyapps.io/Scotus_Clustering/). Finally, deep learning techniques were employed, such as artificial neural networks for classification, word2vec, and doc2vec, to help predict/classify case topics and make word group associations.

1 Network and NLP Data Storage

The research group decided to apply NLP[2] techniques on the opinion text files of our existing citation network of the Supreme Court of the United States (SCOTUS), since this has been largely untouched in the past. As a refresher, the network[3] is comprised of nodes and directed

---

[1] Michael Kim, Scott Garcia, James Jushchuk, and Ethan Koch are co-authors
[2] Natural language processing--computational techniques to process through large corpora of text **[1]**
[3] A "network" may also be referred to as a "graph"

edges[4]. The nodes represent the SCOTUS cases and the directed edges represent the citation relationship between two cases (i.e. if the edge points from node 1 to node 2, this is equivalent to case 1 citing case 2). Furthermore, note that a case cannot cite itself. Therefore, the SCOTUS network is a directed, acyclic graph (DAG[5]). No edge weights[6] were assigned.

Since the last iteration, the network has been further cleaned, reducing its size down to 24,724 cases and 232,999 edges (previously 33,248 cases and 250,449 edges in Fall 2016). Anyone can download the network data (*scotus_network.graphml*, *edgelist.csv*, *case_metadata.csv*, *cluster & opinion JSON files*) and NLP data (opinion text files, tf-idf matrix[7]) through the file *download_data.ipynb*[8] in the research group's public git repository, https://github.com/idc9/law-net.

1.1   Network Data

- *scotus_network.graphml*--SCOTUS network in GraphML format that can be loaded through network analysis packages, such as igraph or NetworkX
    - Nodes = cases, also known as "opinions"
    - Edges (directed) = citations
- *edgelist.csv*--all the citation relationships, where each citation represents an edge from a citing case to a cited case
- *case_metadata.csv*--contains information on each SCOTUS case
    - 'id'--case id as denoted by the supreme court database (SCDB)
    - 'date'--date of case
    - 'court'--jurisdiction name, which is just SCOTUS at the moment
        - (*download_data.ipynb* allows the user to work with other jurisdiction subnetworks--see https://www.courtlistener.com/api/jurisdictions/ for list of all jurisdictions available)
    - 'term'--"identifies the term in which the Court handed down its decision. For cases argued in one term and reargued and decided in the next, term indicates the latter (terms start in first Monday in October)" **[3]**
    - 'issueArea' **[3]**
        - 1--Criminal Procedure (issues 10010-10600)
        - 2--Civil Rights (issues 20010-20410)
        - 3--First Amendment (issues 30010-30020)

---

[4] Directed edges--unliked undirected edges, the edges indicate a directional relationship between two nodes **[2]**

[5] Directed, Acyclic Graph (DAG)--edges are directed and nodes are acyclic, meaning a node cannot have an edge pointing to itself **[2]**

[6] Edge weights--numerical values assigned to individual edges, i.e. length of a road in a road network **[2]**

[7] Term frequency: inverse document frequency--determines corresponding importance of each lemma (singular or root form of word) to each text's context **[1]**

[8] Data storage work done by Iain Carmichael

- - - - 4--Due Process (issues 40010-40070)
      - 5--Privacy (issues 50010-50040)
      - 6--Attorneys/Governmental Fees/Compensation (issues 60010-60040)
      - 7--Unions (issues 70010-70210)
      - 8--Economic Activity (issues 80010-80350)
      - 9--Judicial Power (issues 90010-90520)
      - 10--Federalism (issues 100010-100130)
      - 11--Interstate Relations (issues 110010-110030)
      - 12--Federal Taxation (issues 120010-120040)
      - 13--Miscellaneous (issues 130010-130020)
      - 14--Private Action (issues 140010-140080)
    - 'decisionDirection'--the procedure of determining decision direction is detailed under the SCDB of Washington University Law (St. Louis) **[3]**
      - 1--Conservative
      - 2--Liberal
      - 3--Unspecified
    - 'majVotes'--number of justices voting in majority **[3]**
    - 'minVotes'--number of justices voting in dissent **[3]**
  - *cluster and opinion JSON files*--contains variety of information, such as the case metadata and opinion text files

1.2 NLP Data: Term Frequency - Inverse Document Frequency Matrix (tf-idf matrix)

- tf-idf matrix is acquired by processing through the opinion text files and using scikit's TfidfVectorizer method **[4]**
- Sparse matrix[9] (27,885×567,570) with 20,817,470 nonzero elements
  - Rows: correspond to id's of opinion texts of SCOTUS cases
  - Columns: correspond to unique lemmas[10]
    - filtered by ignoring stop words[11] and using stemming[12] and tokenization[13] processes
  - Elements: tf-idf values

---

[9] Due to the sheer volume of text documents, much of the individual documents will not contain a significant portion of the 567,570 lemmas, explaining the sparse nature of the matrix (many entries are zero)

[10] Lemma--singular or root form of a word **[1]**

[11] Stop words--filtered words from NLP on text data; usually extremely common words,such as pronouns, conjunctions, and auxiliary verbs (i.e. "a", "is", "the", "she", "but") **[1]**

[12] Stemming--standardizing text by reducing words to its common base form (commonly by chopping off ends of words) **[5]**

[13] Tokenization-- breaking up a text document into pieces, possibly discarding some characters, such as punctuation **[5]**

■ Number of times the word appears in the opinion divided by the number of times the word appears in the corpus

2   Other forms of NLP Data

2.1   NLP Data: Bag-of-Words Matrix (bow matrix)

- Bag-of-words matrix[14] is acquired by processing through the opinion text files using scikit's CountVectorizer method **[4]**
- Sparse matrix (27,885×567,570) with nonzero 20,817,470 elements
  - ○ Same rows and columns structure as tf-idf matrix
  - ○ Elements: number of times the word appears in the opinion

2.2   NLP Data: Singular Value Decomposition (SVD) of tf-idf Matrix

The tf-idf matrix is clearly very large in size (27,885×567,570) and will present computational and memory problems when executing future clustering algorithms (see section 3 of paper). Therefore, techniques to reduce the dimensions of the matrix are needed. One such technique is called singular value decomposition (SVD), which produces a low-dimensional representation of the tf-idf matrix by decomposing it into three parts.

Murphy **[6]** explains that an original matrix, $\mathbf{X}$ ($N{\times}D$) can be decomposed into matrices $\mathbf{U}$ ($N{\times}N$) and $\mathbf{V}$ ($D{\times}D$) and diagonal matrix[15] $\mathbf{S}$ ($N{\times}D$). Namely,

$$\mathbf{X} = \mathbf{USV^T}$$

Characteristics of the three matrices:
- Columns of $\mathbf{U}$ are orthonormal ($\mathbf{U^TU} = \mathbf{I_N}$)
- Rows and columns of $\mathbf{V}$ are orthonormal ($\mathbf{V^TV} = \mathbf{VV^T} = \mathbf{I_D}$)
- $\mathbf{S}$ is a diagonal matrix that contains $\min(N,D)$ singular values[16] of $\mathbf{X}$
- Left singular vectors are contained in the columns of $\mathbf{U}$
- Right singular vectors are contained in the columns of $\mathbf{V}$.

Murphy derives the following relationship between eigenvectors/eigenvalues[17] and the singular vectors:

---

[14] Bag-of-words--text documents are considered as collection or bag of words, disregarding word order or grammar but retaining the original word count **[1]**

[15] Diagonal matrix--only the diagonal elements of the matrix are non-zero; the rest of the elements in the matrix are zero **[7]**

[16] Singular values (of square matrix $\mathbf{X}$)--the square roots of the eigenvalues of $\mathbf{X^HX}$, where $\mathbf{X^H}$ is the conjugate transpose of $\mathbf{X}$ **[7]**

$$\mathbf{X} = \mathbf{USV^T}$$
$$\mathbf{X^TX} = \mathbf{(USV^T)^T(USV^T)} = \mathbf{VS^TU^TUSV^T} = \mathbf{VS^TISV^T} = \mathbf{V(S^TS)V^T} = \mathbf{VS^2V^T}$$
$$\mathbf{(X^TX)V} = \mathbf{VS^2}$$

Results:
- right singular vectors, $\mathbf{V}$ = eigenvectors of $\mathbf{X^TX}$
- left singular vectors, $\mathbf{U}$ = eigenvectors of $\mathbf{XX^T}$
- $\mathbf{S^2}$ = eigenvalues of $\mathbf{XX^T}$ = eigenvalues of $\mathbf{X^TX}$

For the tf-idf matrix, truncated SVD **[6]** (implemented in scikit) is applied to allow for dimensionality reduction to rank[18] R, i.e. reducing the dimension from (27,885×567,570) to (27,885×R). R = 500 is chosen, making the SVD matrix have dimension (27,885×500) with 13,942,500 elements. The SVD matrix was returned as a dense matrix.

2.3   NLP Data: Non-Negative Matrix Factorization (NMF) of tf-idf Matrix

With similar reasoning as SVD, non-negative matrix factorization (NMF) is another method in reducing the dimensions of the tf-idf matrix for computational and memory reasons. NMF produces a low-dimensional representation of the tf-idf matrix by decomposing it into two, non-negative matrices. This is possible due the non-negative nature of the tf-idf matrix (elements and components are non-negative).

Overview of NMF is presented, but much more details are in Lee and Seung (2001) **[8]**: Non-negative matrix $\mathbf{X}$ ($N \times D$) can be decomposed into non-negative matrices $\mathbf{W}$ ($N \times r$) and $\mathbf{H}$ ($r \times D$), where r <= max(N,D). Namely,

$$\mathbf{X} \approx \mathbf{WH}$$

The aim is to do alternating minimizations of $\mathbf{W}$ and $\mathbf{H}$:

$$\mathbf{W}_{ik} \leftarrow \mathbf{W}_{ik} \frac{\sum_{j=1}^{D} \mathbf{H}_{kj}\mathbf{X}_{ij}/(\mathbf{WH})_{ij}}{\sum_{j=1}^{D} \mathbf{H}_{kj}}$$
$$\mathbf{H}_{kj} \leftarrow \mathbf{H}_{kj} \frac{\sum_{i=1}^{N} \mathbf{W}_{ik}\mathbf{X}_{ij}/(\mathbf{WH})_{ij}}{\sum_{i=1}^{N} \mathbf{W}_{ik}}$$

---

[17] For linear transformation $\mathbf{T}$, if there is a vector $\mathbf{v}$ such that $\mathbf{Tv} = \lambda\mathbf{v}$ for scalar λ, then λ = eigenvalue($\mathbf{T}$) and $\mathbf{v}$ = eigenvector($\mathbf{T}$) **[7]**

[18] Rank (of a matrix)--the number of linearly independent rows or columns of the matrix (i.e. each column cannot be computed using linear combination of the other columns) **[7]**

to minimize the cost/loss function $L(\mathbf{W},\mathbf{H})$. Namely,

$$L(\mathbf{W}, \mathbf{H}) = \sum_{i=1}^{N} \sum_{j=1}^{D} \mathbf{W}_{ij} log \frac{\mathbf{W}_{ij}}{\mathbf{H}_{ij}} - \mathbf{W}_{ij} + \mathbf{H}_{ij}$$

NMF (implemented in scikit) is applied on the tf-idf matrix to allow for dimensionality reduction to rank R, i.e. reducing the dimension from (27,885×567,570) to (27,885×R). R=250 is chosen, making the NMF matrix have dimension (27,885×250) with 2,335,004 elements. R was chosen to be 250 for the NMF matrix due to time constraints--namely, the computational time for NMF with rank 500 was too high. The NMF matrix was returned as a dense matrix.

3   Clustering on Data

There are five data types that were covered. One was the network data of the SCOTUS citation network and four were NLP data, all concerning the opinion texts of the SCOTUS network.
- Network data
  - *scotus_network.graphml*
- NLP data
  - Term frequency-inverse document frequency matrix (tf-idf matrix)
  - Bag-of-words matrix (bow matrix)
  - Singular value decomposition of tf-idf matrix (SVD)
  - Non-negative matrix factorization of tf-idf matrix (NMF)

For the one network data, there will be two clustering[19] methods to be performed:
- Modularity (mod)
- Walktrap (wt)

For each of the four NLP data, there will be three clustering methods to be performed:
- K-means Clustering (KM)
- Gaussian Mixture Models (GMM)
- Hierarchical Clustering (HC)

Each opinion is assigned to a cluster (group of opinions) when using one of the clustering methods. The clustering work was done under four ipython notebooks *cluster_work_...ipynb*

---

[19] Cluster of graph--subgraph of original graph that has a defining characteristic or shared trait among its vertices **[2]**; for our purpose, each opinion id is assigned to a cluster (group of opinions) using one of the clustering methods

(view https://github.com/idc9/law-net/tree/michael2). These clusters of opinions will later be analyzed using the "summarize cluster" functions in section 4.

3.1   Modularity Clustering on SCOTUS Citation Network

The modularity of the network measures how "tight" the clusters in that network and is a popular benchmark measure for how good the clusters were assigned to each node. Assuming some pre-assigned modules/groups to each node, modularity captures the concentrational difference between the actual number of edges in the modules and the supposed number of random edges that would've fell in those modules **[9]**. Assuming no edge weights in graph, modularity of the graph is defined as,

$$Q = \frac{1}{4m} \sum_{i,j} (\mathbf{A}_{ij} - \frac{k_i k_j}{2m}) s_i s_j$$

where
- Q = modularity
- m = number of edges
- $\mathbf{A}_{ij}$ = ij^{th} element of adjacency matrix[20] $\mathbf{A}$
- k_i = degree of node i
- k_j = degree of node j
- $k_i k_j / 2m$ = probability of random edge between nodes i and j
- $s_i s_j$ = 1 if nodes i and j are in the same group/module, 0 otherwise

igraph's implementation of modularity clustering is used on the SCOTUS citation network (*scotus_network.graphml*), which is only a slight modification of the above, where the normalization factor is 1/2m, rather than 1/4m **[10]**.

The modularity clustering algorithm implemented in igraph takes a bottom-up, greedy heuristic which attempts the following **[10]**:
1. Assign random, separate clusters (modules) to each node
2. Merge a node into a cluster such that modularity score of the graph is maximized
3. Repeat step 2 until merging cannot increase current modularity score

The math for the greedy algorithm is outlined under Newman (2006) **[9]**.

---

[20] <u>Adjacency matrix</u>--matrix representing whether two vertices are adjacent in the graph, i.e. for an acyclic graph with no edge weights, if $A_{ij}=0$, then node *i* is NOT adjacent/connected to node *j* (so diagonal elements=0); if $A_{ij}=1$, then node *i* IS adjacent/connected to node *j* **[2]**

Considering modularity score calculations disregard edge direction, modularity clustering was performed on the undirected, largest connected component[21] form of the SCOTUS network, which only loses 223 out of 2,332,222 edges from the original SCOTUS DAG.

For the modularity clustering, igraph generally defaults into deciding the best number of clusters for the graph rather than the user pre-specifying it, mainly due to the greedy optimization approach. 126 clusters were assigned at the end of algorithm.

3.2   Walktrap Clustering on SCOTUS Citation Network

Pons and Latapy (2005) **[11]** explains that the premise behind walktrap clustering is how random walks[22] can help define communities[23]. This is because most random walks will stay within the same community with high probability, due to the community's dense structure. Walktrap clustering first attempts to select which communities to merge using the distances between communities from random walks in a bottom-up, greedy fashion, much like the modularity clustering algorithm (more specifically Ward's method is used for community merges, a form of agglomerative hierarchical clustering, which is further discussed in section 3.5). Then, it decides where to cut the dendrogram[24] of communities to get the clusters. This is the same as determining the "best" partition that captures the community structure well (the partition with maximum modularity score). The sets of this "best" partition are the walktrap clusters.

An overview of the algorithm and few of its maths behind walktrap clustering is as follows **[11]**:
1. Start with a single partition $\mathcal{P}_1$ that holds all nodes
2. Assign random, separate communities to each node (*n* communities for *n* nodes)
3. Compute distances between every adjacent community by using random walk on each node (can be modeled as Markov chain process)
   a. Distance between two nodes *i* and *j*
   $$r_{ij} = \sqrt{\sum_{k=1}^{n}(P_{ik}^t - P_{jk}^t)^2/d(k)} = \left\| \mathbf{D}^{-P_{i*}^t/2}\mathbf{D}^{-P_{j*}^t/2} \right\|$$
      i. P_{ij}^t is the probability to go from node *i* to node *j* in *t* steps
      ii. $d(i)$ is the degree of node *i*

---

[21] <u>Largest connected component of graph</u>--subgraph of original graph, where one node will always be connected to another node **[2]**
[22] <u>Random walk on graphs</u>--at each unit of time, a "walk" (path) is taken from a node to one of its neighbors (the neighbor is randomly chosen) **[6]**
[23] <u>Communities (of a graph)</u>--different from clusters of graph, a community is a group of nodes that are densely connected to one another **[2]**
[24] <u>Dendrogram</u>--binary tree where leaves=nodes, branches=communities, and stems=community merges (idea: communities of communities) **[11]**

iii. $\|\cdot\|$ is the euclidean norm[25] of $\mathbb{R}^n$

iv. $\mathbf{D}$ is the diagonal matrix ($n \times n$) of the node degrees

v. P_{i*}^t is probability distribution to go from node i to all of its neighbors in t steps

b. Distance between two communities, C_1 and C_2

$$r_{C_1 C_2} = \left\| \mathbf{D}^{-P_{C_1*}^t/2} \mathbf{D}^{-P_{C_2*}^t/2} \right\| = \sqrt{\sum_{k=1}^{n} (P_{C_1 k}^t - P_{C_2 k}^t)^2 / d(k)}$$

i. $P_{C_j}^t = \dfrac{\sum_{i \in C} P_{ij}^t}{|C|}$ is the probability to go from community C to node j in t steps

4. For each step k: Choose two communities C_1 and C_2 in $\mathcal{P}_k$ to merge into a new community $C_3 = C_1 \cup C_2$ based on Ward's method (the merge of two communities that reduces variation $\triangle\sigma(C_1, C_2)$ of σ)

a. Mean of squared distances between each node and its respective community

$$\sigma_k = \frac{1}{n} \sum_{C \in \mathcal{P}_k} \sum_{i \in C} r_{iC}^2$$

b. Variation $\triangle\sigma(C_1, C_2)$ of σ if C_1 and C_2 were merged as $C_3 = C_1 \cup C_2$

$$\triangle\sigma(C_1 C_2) = \frac{1}{n} \left(\sum_{i \in C_3} r_{iC_3}^2 - \sum_{i \in C_1} r_{iC_1}^2 - \sum_{i \in C_2} r_{iC_2}^2 \right) = \frac{1}{n} \frac{|C_1||C_2|}{|C_1| + |C_2|} r_{C_1 C_2}^2$$

5. Create new partition $\mathcal{P}_{k+1} = (\mathcal{P}_k \backslash \{C_1, C_2\}) \cup \{C_3\}$

6. Repeat step 3

7. Repeat steps 4-6 *k-1* times to get a hierarchical structure of communities (the dendrogram of communities)

8. Determine the height to cut the dendrogram (choose the "best" partition, the partition with the highest modularity score)

a. Modularity of each partition (this definition of modularity is simpler than before)

$$Q(\mathcal{P}) = \sum_{C \in \mathcal{P}} e_C - a_C^2$$

i. e_C is the fraction of edges inside the community

ii. a_C is the fraction of edges with at least one edge in the community (square root of the fraction of random edges that would fall into the community)

The sets of the "best" partition from step 8 are the walktrap clusters

---

[25] Euclidean norm--the euclidean norm of vector **x** would be the square root of the sum of its squared elements **[12]**

Like modularity clustering, walktrap clustering was also performed on the undirected, largest connected component form of the SCOTUS network. This is because the SCOTUS network is originally directed and acyclic, meaning random walks may end abruptly at nodes with no outgoing edge. This implies that random walks should be performed on undirected graphs for meaningful results. Furthermore, there is consistency of data type used for both modularity clustering and walktrap clustering.

For walktrap clustering, igraph generally defaults into deciding the best number of clusters for the graph rather than the user pre-specifying it, due to the greedy optimization approach of community merges. 2,264 clusters were assigned at the end of algorithm.

3.3   K-Means (KM) Clustering on NLP Data

Sections 3.1 and 3.2 covered two clustering algorithms that were applied on the SCOTUS citation network. The rest of the sections onward (sections 3.3-3.5) will cover clustering algorithms were applied on NLP data (k-means, Gaussian mixture models, hierarchical clustering).

The overall method is to randomly generate K means and assign opinions to the K disjoint[26] clusters with the closest means. This process repeats until the variance among all the clusters is minimized **[13]**.

For illustrative purposes, the tf-idf matrix (27,885×567,570) is used as the NLP dataset for the k-means clustering algorithm described below **[13]**:
1. Choose K, the desired number of distinct, disjoint clusters (clusters $1 \rightarrow K$)
2. For initialization,randomly assign each opinion to a cluster
    a. Note: each opinion can be thought of as a row vector from the tf-idf matrix (recall that in a tf-idf matrix, each column corresponds to a lemma and the elements are tf-idf values)
3. For each cluster, computer the cluster centroid
    a. Note: Cluster k with n opinions can be viewed as a sub-matrix of the tf-idf matrix (n×567,570)
    b. Note: Centroid of cluster k is the mean vector of the sub-matrix's row vectors
4. Assign each opinion to a cluster with the "closest" centroid (centroid with minimum euclidean distance)
    a. Euclidean distance of cluster k:

---

[26] <u>Disjoint</u>--clusters *A* and *B* are disjoint if they do not share an observation in common; also known as "non-overlapping" **[13]**

$$\sqrt{\sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2}$$

- x_{ij} is tf-idf value of lemma j of opinion i
- $x_{i'j}$ is tf-idf value lemma j of centroid i'

5. Iterate the previous two steps until total within-cluster variation cannot be further minimized

   a. Total within-cluster variation:

$$\sum_{k=1}^{K} W(C_k)$$

   b. , where W(C$_k$) is the Squared euclidean distance of cluster k:

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2$$

- $|C_k|$ number of opinions in cluster k

K-means clustering for K = 10, 100, 1000 clusters was attempted on all four of the NLP datasets (tf-idf matrix, bow matrix, SVD matrix, NMF matrix). Unfortunately, k-means with K = 1000 clusters did not work for the tf-idf matrix and bow matrix due to a memory error. This is most likely due to the fact that both matrices are large and sparse, and doing k-means with K = 1000 on a CSR[27] matrix format may be too memory intensive.

3.4   Gaussian Mixture Models (GMM) Clustering on NLP Data

Gaussian mixture model can be thought of as a slight alternative to k-means that makes probabilistic (not deterministic) assignments of points to K mixture components (rather than K clusters) **[14]**. It assumes that a mixture of Gaussian distributions with unknown parameters can generate data points by using the covariance structure of the data and centers of latent Gaussian distributions **[4]**.

The Gaussian mixture model is defined as **[14]**:

$$f(x) = \sum_{m=1}^{M} \alpha_m * \phi(x|\mu_m, \Sigma_m)$$

- α_m are the "weights" or "mixing coefficients" of Gaussian density m, where

$$\sum_m \alpha_m = 1$$

---

[27] Compressed Sparse Row matrix

- $\phi(\cdot)$ is Gaussian density with mean μ_m and variance Σ_m

Given a set of observations, maximum likelihood estimation (MLE) estimates unknown parameters of a distribution, such that the distribution has maximum likelihood to generate the given observations. In other words, given iid *n* observations, *x₁, x₂, …, xₙ*, MLE is a maximization problem of the likelihood function, $\mathcal{L}$, which is in the general form **[14]**:

$$\mathcal{L}(\theta|x_1, x_2, ..., x_n) = f(x_1, x_2, ..., x_n|\theta) = \prod_{i=1}^{n} f(x_i|\theta)$$

- $f(x_1, x_2, ..., x_n|\theta)$ is the joint density function

For numerical purposes, the log-likelihood function (log transformation of $\mathcal{L}$) is often used:

$$log[\mathcal{L}(\theta|x_1, x_2, ..., x_n)] = \sum_{i=1}^{n} log[f(x_i|\theta)]$$

However, MLE alone is numerically complicated for most mixture models, as is the case for using it to estimate the parameters of GMM. The Expectation-Maximization (EM) algorithm is employed to simplify the MLE problem on the mixture of Gaussians **[14]**. The math below is taken from *Elements of Statistical Learning* (*ESL*) by Hastie et. al. **[14]** and illustrates the EM algorithm nicely for a two-component Gaussian mixture model:

Model *Y* as mixture of two Gaussian distributions, *Y₁* and *Y₂*:
$$Y_1 \sim N(\mu_1, \sigma_1^2) \text{ is model 1}$$
$$Y_2 \sim N(\mu_2.\sigma_2^2) \text{ is model 2}$$
$$Y = (1 - \Delta) \cdot Y_1 + \Delta \cdot Y_2$$
- $\Delta \in \{0, 1\}$

Let:
- $P(\Delta = 1) = \pi$
- $\theta = (\mu, \sigma^2)$ is Gaussian parameter with mean μ and variance σ^2

Density of Y:
$$g_Y(y|\boldsymbol{\theta}) = (1 - \pi)\phi_{\theta_1}(y) + \pi\phi_{\theta_2}(y)$$
- $\boldsymbol{\theta} = (\pi, \theta_1, \theta_2) = (\pi, \mu_1, \sigma_1^2, \mu_2, \sigma_2^2)$

Log-likelihood for observed data **Z** (*n* training cases)--faces numerical problem:

$$\mathcal{L}(\boldsymbol{\theta}|\boldsymbol{Z}) = \sum_{i=1}^{n} log[(1-\pi)\phi_{\theta_1}(y_i) + \pi\phi_{\theta_2}(y_i)]$$

Log-likelihood for observed data $\boldsymbol{Z}$ and unobserved latent variables $\boldsymbol{\Delta}$--fixes numerical problem:

$$\mathcal{L}_0(\boldsymbol{\theta}|\boldsymbol{Z}, \boldsymbol{\Delta}) = \sum_{i=1}^{n}[(1-\Delta_i)log\phi_{\theta_1}(y_i) + \Delta_i log\phi_{\theta_2}(y_i)]$$
$$+ \sum_{i=1}^{n}[(1-\Delta_i)log(1-\pi) + \Delta_i log\pi]$$

- $\Delta_i \in \{0, 1\}$ is the unobserved latent variable, where
  - If $\Delta_i = 1$, then Y_i is from model 2
    - Note: Maximum likelihood estimates of μ_2 and σ_2^2 = sample mean and variance for data with $\Delta_i = 1$
  - If $\Delta_i = 0$, then Y_i is from model 1
    - Note: Maximum likelihood estimates of μ_1 and σ_1^2 = sample mean and variance for data with $\Delta_i = 0$
- Note: π can be estimated as the proportion of Δ_i that is 1

From the above result, we get expectation of $\Delta_i \Leftrightarrow$ "responsibility" of model 2:
$$\gamma_i(\boldsymbol{\theta}) = \mathrm{E}(\Delta_i|\boldsymbol{\theta}, \boldsymbol{Z}) = P(\Delta_i = 1|\boldsymbol{\theta}, \boldsymbol{Z})$$

EM algorithm for two-component Gaussian mixture model:
1. Initial guesses for parameters: $\hat{\mu}_1, \hat{\sigma}_1^2, \hat{\mu}_2, \hat{\sigma}_2^2, \hat{\pi}$
   a. $\hat{\mu}_1$ and $\hat{\mu}_2$ can be guessed by choosing two random y_i's.
   b. $\hat{\sigma}_1^2$ and $\hat{\sigma}_2^2$ can be equal to overall sample variance:
   $$\sum_{i=1}^{n} \frac{(y_i - \bar{y})^2}{n}$$
   c. $\hat{\pi}$ can be guessed as 0.5
2. Expectation step: after soft assigning each observation to model 1 or 2, compute responsibilities,
   $$\hat{\gamma}_i = \frac{\hat{\pi}\phi_{\hat{\theta}_2}(y_i)}{(1-\hat{\pi})\phi_{\hat{\theta}_1}y_i + \hat{\pi}\phi_{\hat{\theta}_2}(y_i)}, i = 1, 2, ..., N$$
3. Maximization step: update parameter estimates $\Leftrightarrow$ compute weighted means and variances and mixing probability
   $$\hat{\mu}_1 = \frac{\sum_{i=1}^{n}(1-\hat{\gamma}_i)y_i}{\sum_{i=1}^{n}(1-\hat{\gamma}_i)}$$
   $$\hat{\mu}_2 = \frac{\sum_{i=1}^{n}\hat{\gamma}_i y_i}{\sum_{i=1}^{n}\hat{\gamma}_i}$$

$$\hat{\sigma}_1^2 = \frac{\sum_{i=1}^{n}(1 - \hat{\gamma}_i)(y_i - \hat{\mu}_1)^2}{\sum_{i=1}^{n}(1 - \hat{\gamma}_i)}$$

$$\hat{\sigma}_2^2 = \frac{\sum_{i=1}^{n}\hat{\gamma}_i(y_i - \hat{\mu}_2)^2}{\sum_{i=1}^{n}\hat{\gamma}_i}$$

$$\hat{\pi} = \sum_{i=1}^{n}\frac{\hat{\gamma}_i}{n}$$

4. Iterate steps 2 and 3 until convergence (local maximum of likelihood)

GMM clustering for K = 10, 100, 1000 mixture components was attempted on all four of the NLP datasets (tf-idf matrix, bow matrix, SVD matrix, NMF matrix). Unfortunately, GMM for any number of K mixture components did not work on the tf-idf matrix and bow matrix due to scikit's implementation of GMM requiring a dense matrix form. Converting the large, sparse matrices into its respective dense form causes memory error.

3.5 Hierarchical (HC) Clustering on NLP Data

Agglomerative hierarchical clustering is also applied on the four NLP datasets. This hierarchical clustering method follows a straightforward algorithm that essentially attempts to merge pairs of clusters that are most "similar" to one another. First, it starts off by treating each observation as a cluster. Two clusters that are most "similar" to each other are merged at each step. A dendrogram (tree-like data structure) is built, where each leaf represents an observation (i.e. SCOTUS opinion) and each stem represents where two clusters have merged during the algorithm [14].

Once a dendrogram is built, one can decide the number of clusters they want. Namely, one decides the height at which the dendrogram is cut. After the cut, the stems at the very top represent the clusters, and the leaves at the bottom respective to each stem are the observations for each cluster. Hence, this form of hierarchical clustering is agglomerative, as the dendrogram of clusters is built, starting from the ground-up (also known as bottom-up approach in algorithm language). The opposite of agglomerative clustering is divisive clustering, where the algorithm starts off with a single cluster holding all observations which is recursively divided into two children clusters in a top-down approach. However, this is not used for our purposes, as agglomerative clustering is much more extensively studied and used generally [14]. Furthermore, divisive clustering is not implemented in scikit as of yet.

Previously, it was mentioned that clusters were merged that were most "similar" to one another. Another way to think of this is that merges occur between pairs of clusters that are least "dissimilar" to one another. We define two terms for our purposes [13]:
- Dissimilarity--measure used between a pair of observations to describe how dissimilar they are; generally some form of distance measure between two observations.

- Euclidean distance is used, as this is the most commonly used and is also the default option under scikit's implementation of agglomerative clustering **[4]**.
- Linkage--dissimilarity measure between two groups of observations (pairwise intercluster dissimilarity measure)
  - Ward linkage is used, as this is the the default option under scikit's implementation of agglomerative clustering and works well with euclidean dissimilarity; the aim is to find the two clusters that would have minimum variance when merged **[4]**

Hierarchical clustering for $K = 10, 100, 1000$ clusters was attempted on all four of the NLP datasets (tf-idf matrix, bow matrix, SVD matrix, NMF matrix). Unfortunately, hierarchical clustering for any number of K clusters did not work on the tf-idf matrix and bow matrix due to scikit's implementation of hierarchical clustering requiring a dense matrix form. Converting the large, sparse matrices into its respective dense form causes memory error.

4   Summarizing the Clusters

A tree is listed below that shows the clustering methods that worked in section 3. Each leaf can be seen as the set of clusters acquired from a method (i.e. the third leaf from the left is the set of clusters from k-means clustering on tf-idf for $K = 10$):
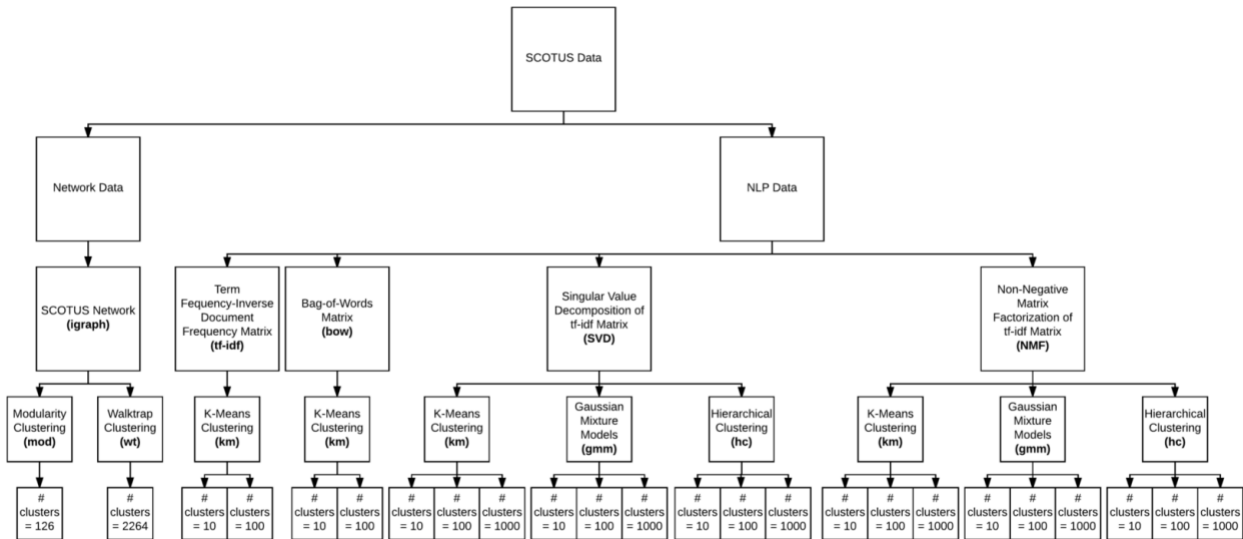
Figure 1: Tree of Working Clustering Methods

The aim in section 4 is to explain four functions we implemented in python that can "summarize" any cluster by using the tf-idf matrix. The user generally specifies the set of opinions (cluster), number of words to return, tf-idf matrix, dictionary that relates row index to opinion id, and a list of vocab pertaining to the column indices of the tf-idf matrix.

- The last three parameters can be acquired from the *download_data.ipynb* in the research group's public git repository, https://github.com/idc9/law-net.
- The four functions can be found under https://github.com/idc9/law-net/blob/michael2/code/summarize_clusters.py.
- These functions were executed in twenty four ipython notebooks, *summarize_...ipynb* (view https://github.com/idc9/law-net/tree/michael2), and the results were saved as CSV files for the later development of shiny app in R (section 5).

4.1   Summarize Cluster Function #1

This function "summarizes" a cluster by returning a set of distinct words in the cluster with the highest tf-idf scores. 1000 words were chosen to be returned.

4.2   Summarize Cluster Function #2

This function "summarizes" a cluster by returning the words with highest tf-idf scores from the mean tf-idf vector of the cluster. The mean tf-idf vector of the cluster is defined as the mean of the row vectors from the tf-idf matrix that correspond to the opinions of the cluster. 1000 words were chosen to be returned.

4.3   Summarize Cluster Function #3

This function "summarizes" a cluster by returning the words with highest tf-idf scores from the difference of the mean tf-idf vector of the cluster and the mean tf-idf vector of the complement of the cluster. The complement of the cluster is defined as the set of opinions not in the cluster. 1000 words were chosen to be returned.

4.4   Summarize Cluster Function #4

Unlike the previous three summarize cluster functions, function #4 "summarizes" a cluster by returning the opinion in the cluster that is "closest" to the mean tf-idf vector of the cluster. Here, each opinion in the cluster is treated as their respective row vector from the tf-idf matrix, and every euclidean distance between each opinion's tf-idf vector and the mean tf-idf vector is measured. The opinion with a tf-idf vector with minimum euclidean distance to the mean tf-idf vector is chosen to be the most "relevant" opinion of the cluster.

5   Showcasing Results in Shiny App

The purpose of the work done in previous sections was to build an interactive app for users (particularly legal practitioners) to easily view and find potential relationships between a set of

opinions and its top words + most "relevant" opinion. The shiny app in R makes this easy to accomplish. We can now show word contextualization of legal court cases through the use of NLP and statistical techniques: https://scottgarcia.shinyapps.io/Scotus_Clustering/. The hope is that this could lead to future collaborations that hypothesize and possibly prove correlations between clustering methods, NLP data structure, and court cases.

Exploring the shiny app may be daunting at first, but much of its interface was built for the hope to be user-friendly. The subsections below will attempt to explain shiny app's features. There are three tabs to be explored, "Individual Cases," "Case Metadata Summaries", and "Text Summaries."

We hope to implement more features in the future, while also keeping balance between quantity vs. quality. For instance, there is a fine line between presenting the right amount of information and presenting too much information which creates unnecessary visual noise. Thus, tailoring to the needs of shiny app users may mean adding more features, removing unwanted information, and/or improving the layout of the graphic user interface.

5.1 Choosing the Cluster

For any of the three tabs, one must choose a cluster to analyze. There are three drop-down menus on the left for this purpose. Most of these are self explanatory, but here is a quick overview:
- "Choose Data Representation"--see third row of figure 1
- "Choose Clustering Method"--see fourth and fifth rows of figure 1
- "Choose Cluster"--in descending order from biggest cluster (most number of opinions) to smallest (least number of opinions)

5.2 "Individual Cases" Tab

The first tab, "Individual Cases," attempts to simply present a large table holding all the metadata for a cluster (see section 1.1--*case_metadata.csv*). One can view the 'id', 'date', 'court, name', 'judges', 'scdb_id', 'term', 'issueArea', 'decisionDirection', 'majVotes', and 'minVotes' for each opinion in the cluster.

5.3 "Case Metadata Summaries" Tab

The second tab, "Case Metadata Summaries," has a drop-down menu for three miscellaneous information on the cluster:

- "Data Histogram"--shows histogram of opinion date vs. indegree[28] for a cluster; a common purpose is to see when cases were most cited
- "Popular Cases"--same as the "Individual Cases" tab, but lists the metadata in descending order from opinions with highest indegree to lowest indegree
- "Issue Area Summaries"--defines the issueArea numbers in the metadata tables, which are general areas of law an opinion falls under

5.4   "Text Summaries" Tab

The third tab, "Text Summaries," shows the metadata for the most "relevant" opinion of the cluster (section 4.4), as well as the top 1000 words acquired from each of the first three summarize cluster functions (sections 4.1-4.3). It also shows the number of opinions in the cluster at the very top.

6   Introduction to Deep Learning

6.1   Overview

One of the research interests regarding the SCOTUS cases is identifying groupings, classifications, or distinctions between documents. We identified deep learning to be a potentially powerful tool to accomplish these goals, so we began the semester by learning what exactly this buzzword is. After learning about the theory behind deep learning and how to apply it to a general data set like notMNIST using Tensorflow, a package in Python, we began to applying deep learning to the SCOTUS cases by trying to create a classification model which predicts the case topics. From here we moved to learning about and applying word2vec and doc2vec to the SCOTUS text files in order to create word and document embeddings. We believe these embeddings will prove useful to our original goal of identifying groupings, classifications, or distinctions between cases, as well as answer new research questions, such as comparing the use of words in a court-based context to a more general, everyday context.

6.2   Logistic Classification

The focus of the first Udacity lesson was on the function, logistic classifier:

$$\mathbf{WX} + b = \mathbf{Y}$$

- $\mathbf{W}$ is a matrix of weights
- $\mathbf{X}$ is a vector of inputs
- b is the bias term
- $\mathbf{Y}$ is a vector of outputs

---

[28] Indegree--refers to the indegree of a node in a network, which is the number of incoming edges that a node has; in the case of the SCOTUS network, since each node represents an opinion, indegree is the number of times an opinion is cited [2]

This function is useful for classifying input variables into one of a finite number of discrete classes. This equation outputs a score for any given classification (also called logits) **[17]**. These logits are positive real numbers, but the aim of the classification problem is to assign the class that is most likely associated with the input variable. That requires probabilities, so a softmax function is used to convert the logits into probabilities ranging from 0 to 1 **[17]**. Once these probabilities are calculated, one-hot encoding is used to convert them into an indicator function. This process will output a classification, but it is necessary to optimize the weights and bias so that the logistical classification can be as accurate as possible **[17]**. Training input variables with correct classifications are applied to the logistical classification and cross-entropy is used to determine the accuracy of the parameters. During this process (stochastic) gradient is applied to to find the parameters that minimize the cross-entropy loss.

6.3   Softmax

The logits from the initial operation of $\mathbf{WX} + b$ will be positive real numbers. However, it is ideal to convert these values into probabilities, so an indicator function can be derived later on. The operation to convert the logits into probabilities is called the softmax function. The softmax function takes the form **[17]**:

$$S(y_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

6.4   One-hot Encoding

Ideally the classifier assigns probability very close to 1 for the correct label and probabilities very close to 0 for incorrect labels **[17]**. However, cross-entropy use necessitates a true indicator function. One-hot encoding is the way to transform categorical variables into binary variables. The classification with the highest probability is assigned a 1 while all other classifications are assigned a 0.

6.5   Cross-Entropy

This function measures the distance that the current classification predicted by the logistic classifier is from the classification assigned by the training variables. This cross-entropy is used to create a loss function to find the optimal values for $\mathbf{W}$ and b. This loss function is the average cross-entropy over the training set. It can be minimized using (stochastic) gradient descent.

6.6   Gradient Descent

Gradient descent is carried out by taking steps in the negative direction proportional to the gradient at a given point. The process is carried out many times in order to find a local minimum. For numerical stability reasons, we can normalize the input variables to be centered at 0 and scaled to have equal variance **[17]**. Next, $\mathbf{W}$ and b are initialized to some starting values which will converge once the gradient descent is carried out. We initialize b and $\mathbf{W}$ by

either letting them equal zero or randomly choosing the values from a standard Gaussian distribution. While regular gradient descent is simpler than its stochastic counterpart, it is generally difficult to scale to a larger **W** and *b* as well ast to larger datasets **[17]**.

6.7   Stochastic Gradient Descent (SGD)

To handle large datasets, Stochastic Gradient Descent (SGD) is used. Rather than inputting the entire initial dataset, SGD takes a small sample of the total observations and performs gradient descent on that sample. This will only give an approximation of the true minimum of the function, but it will typically be a very good approximation over many iterations (many more than regular gradient descent). So while many iterations are required for SGD, each iteration is only evaluating a small sample **[17]**. This ends up being more efficient than few iterations with large samples. Similar to gradient descent, the mean of the inputs should be centered at 0, with equal, small variances. **W** is set as a random Gaussian distribution and *b* is set to 0.

There are several extensions for SGD. The first is momentum, which is keeping track of the previous gradients in order to identify which general direction iterations are heading towards **[17]**. The second is AdaGrad which is less sensitive to hyper-parameters (initial learning rate, learning rate decay, momentum, batch size, weight initialization), but can tend to be worse than SGD when working with good parameters **[17]**.

Learning Rate Tuning is recognizing how quickly your model learns. Generally it is not the case that just because the models learns quickly that the model is learning well. When the model is not working as well as intended, lowering the learning rate should be the first adjustment.

7   From Logistic Models to Neural Networks

We now move from a logistic classification framework to an explanation of how artificial neural networks, the backbone of deep learning, can be used to accomplish the same goal. An artificial neural network in this case is essentially the composition of linear operations, with a nonlinear function being applied after each operation, or in other words, at each layer. We understand this to have the effect of approximating extremely complex functions through using simple matrix multiplication, addition, and a few nonlinear functions quickly and with great accuracy. After an explanation of some methods to ensure the correctness of a model in this section, we will move onto an application of artificial neural networks on the notMNIST dataset, provided by Udacity.

7.1   Useful Definitions

Deep feedforward networks (also feedforward neural networks or multilayer perceptrons (MLP)) attempt to approximate some classifying function, *f* **[16]**. These are the neural networks we have thus far worked with in our research. It is called feedforward because there are no ways in which the **x** value pass through the function is plugged back into the function to obtain some other result. When they do include feedback connections, these neural networks are called **recurrent neural networks [16]**. There are many different types of neural networks and the specific choice of neural network and architecture is specific to the task of the model.

The length of the chain of functions is called the **depth** of the model **[16]**. The final layer is called the **output layer [16]**. The **hidden layers** are the layers for which the training data does not show what the output should be, we only have *y* which is the classification and final output of what our function should provide **[16]**.

The dimensionality of the hidden layers is called the **width** of a model **[16]**. Vector-valued representation for deep learning was an idea taken from neuroscience.

7.2   Rectified Linear Units

In a linear model we have *K(N+1)* parameters (*N* inputs, *K* outputs). Linear models are limited in that they cannot model other operations well, though they are useful because they are numerically stable and can be computed quickly by new technologies, namely GPUs. Rectified Linear Units (ReLUs) are simple, nonlinear functions that overcome these issues **[16]**. An ReLU is an artificial neural network is an activation function defined by:

$$f(x) = max(0, x)$$

In this way the model can obtain nonlinear approximations through computationally cheap methods **[16]**. This can be used over several "layers" as well, and, in fact, at least one nonlinear function must be used between each layer; otherwise we would have to do linear operations which could simply be combined into one operation. This would reduce it back to being a linear model. However, with these layers, we can now tune another parameter, *H*, which is the number of ReLUs or hidden layers used. Not all artificial neural networks use ReLUs and it is important to note that the design process, which includes this choice, is full of trial and error. What we have so far is a stack of simple operations: matrix multiplication, addition, ReLUs, matrix multiplication, addition, and then the softmax function **[17]**.

7.3   Backpropagation

In order to use SGD with our newly formed artificial neural network, which is desirable for its flexibility, speed, and accuracy, we need to compute its gradient. Backpropagation is the method by which we achieve this. Backpropagation computes the derivative of a complex function that is made up of blocks of simple functions with respect to its weights **[16]**. In our case, these simple functions are ReLUs. In order to run SGD with backpropagation, we must run the forward propagation (compute the output of the function with the given inputs), run the backward propagation (compute the difference between the actual and target values at each layer), use this information to gradients for each of your weights in your model, update your weights with according to the specified learning rates, and finally, repeat this many times until we reach a local optimum **[16]**. One drawback to backpropagation is that it is twice as memory-demanding as forward propagation.

7.4   Improving the Model

In general, this model works well but can be improved. One way to improve the performance of our neural network is to increase the width of the hidden layer (effectively increasing the number of columns of the matrix in the hidden layer). However, it has been shown that it is better to increase the depth (increase the number of layers) of the network instead. This leads to better generalization to new data. Additionally, this may better reflect the hierarchical structure of the data (e.g. [lines, edges (input)] → [eyes, hair, chin (hidden layers)] → [face (output)]) [16]. One drawback of this approach, however, is that while they generalize well when optimized, they can be harder to optimize in the first place.

At this point, our model can produce very impressive results on our given data. However, this introduces the danger of overfitting, which restricts the model's ability to perform well on new data. One strategy to prevent overfitting in neural networks is early termination, wherein we stop training as soon as our validation set performance stops improving (assuming we have partitioned our data into training, validation, and test sets) [16].

An alternative approach is to apply regularization. To do this we add the L_2 norm (euclidan distance) of weights to the loss function in order to penalize for large weights [16]. A similar approach is dropout. In dropout, we randomly delete half of the activation outputs from moving from one layer to the next. This way, we effectively keep the model from relying on any given activation because it may be deleted at any point [16]. These two forms of regularization can, however, add to the computation time in many circumstances, so often we rely on early termination.

7.5   Example: notMNIST Dataset from Udacity

Similar to the popular MNIST dataset, notMNIST contains hundreds of thousands of images of computer-generated letters. The goal is to create a model that takes in an image of a letter and predicts which letter it is. For our example we used only the letters A through J, though the model could easily be extended to include all letters and digits.

We trained our model on 200,000 images of digits, while our validation and test sets both had 10,000 images. In this example we wished to understand to usefulness of artificial neural networks, so we trained both a logistic classification model and an artificial network. Using a logistic classification model with gradient descent, we achieved a validation set accuracy of 75% and a test set accuracy of 82.6%. Using stochastic gradient descent, we improved to 78.8% and 86.6%, respectively. Clearly stochastic gradient descent is superior for our purposes.

We then constructed a 1-layer neural network with a hidden layer size of 1024. Using SGD, we achieved a validation set accuracy of 82.5% and a test set accuracy of 89.7%. This example shows the superiority of an artificial neural network over a logistic model for classification of

images. Similar improvements over traditionally used models have been demonstrated for other tasks as well.

8 Word2Vec and Doc2Vec

8.1 Word2Vec

While our primary aim in exploring deep learning was to improve the classification of court cases, the large scope of Tensorflow allowed us to explore other applications **[18]**. Word2Vec, as described in detail below, allowed us to both visually and arithmetically compare words in relation to each other. We were interested in the relationship between words in the context of legal writing and if the models we train could recognize these special contexts.

Our initial exploration of word2vec involved training a model to a Wikipedia corpus. Following the Udacity lesson on the topic, we used an input of 17,000,000 words to train both a skip-gram and a continuous-bag-of-words model. After many iterations, both models performed well, with CBOW performing slightly better than the skip-gram model with losses of 1.96 and 3.35, respectively. We then adjusted the program to accept input from our legal cases. This new input dataset contained 127,600,000 words. Once this model was trained, we could visually display these vector representation of word (figure 2) using TSNE from the sklearn python package. This visual representation was able to capture the word associations that the model produced. While the absolute position of words is arbitrary in the plot, note that numbers are all grouped relatively close to each other in the bottom-right. Also many legal terms (such as 'defendant', 'circuit', and 'district') are all in the left-center region. We also improved on the program to allow the exploration of 'neighboring' words, which were the vectors that were nearest to each other in this multidimensional space. As an example, the nearest vectors to the vector associated with 'jury' were 'sooner', 'deplored', 'greenbrier', and 'bequeathed'. Better understanding the results of these neighboring words is a goal of future research.
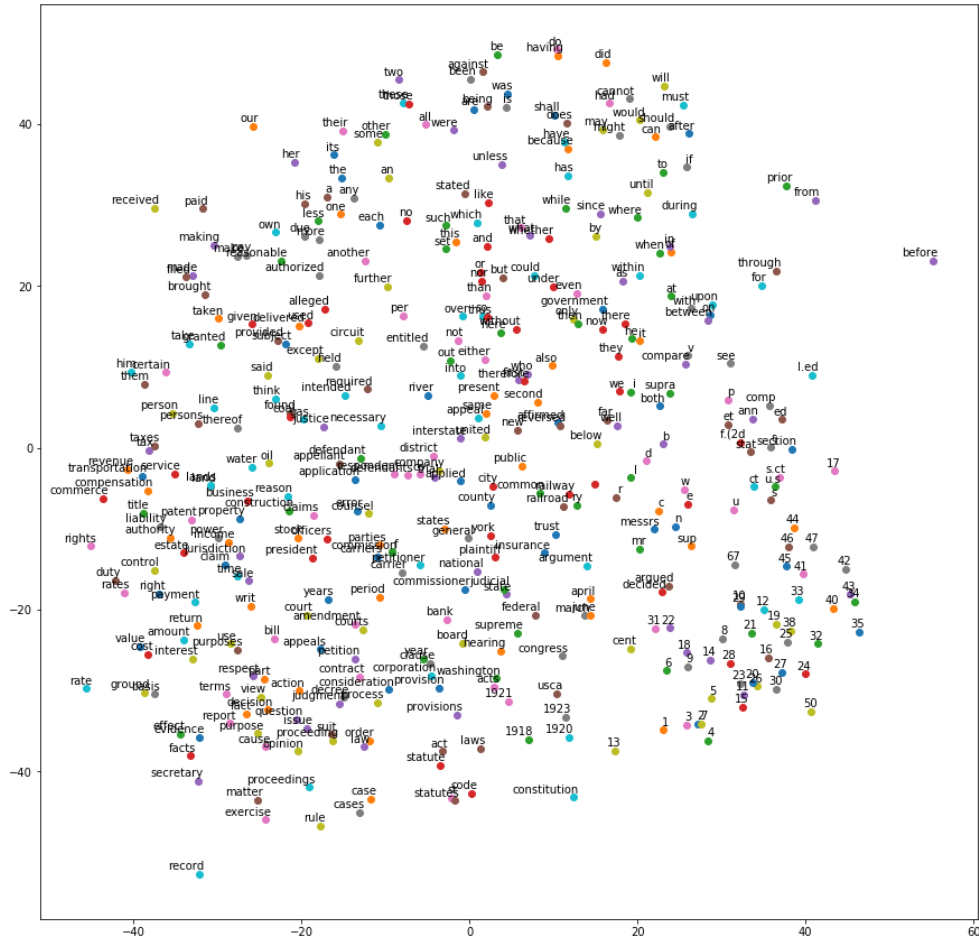
Figure 2: Vector Representation of Words in SCOTUS Text Files

8.2   Doc2Vec

Doc2vec is an extension of word2vec. The implementation we used is the distributed memory model. In this model, we train by predicting the next word from a given context. This context is chosen randomly within a given document and the ID for the chosen document is remembered in the model as being associated with that prediction **[19]**. Eventually this model will learn the semantics of words in a given document and turns this into a document vector **[19]**. We used the Python package Gensim to train and obtain vectors for each of the SCOTUS case text files. These vectors can be used for clustering, classification, and more. We learned doc2vec at the end of the semester and have not yet incorporated it to a very large extent through these methods but are hopeful about the results it may produce in the future. As an example of its usefulness, we used the vectors to graph the similarity of all documents to a particular document over time.
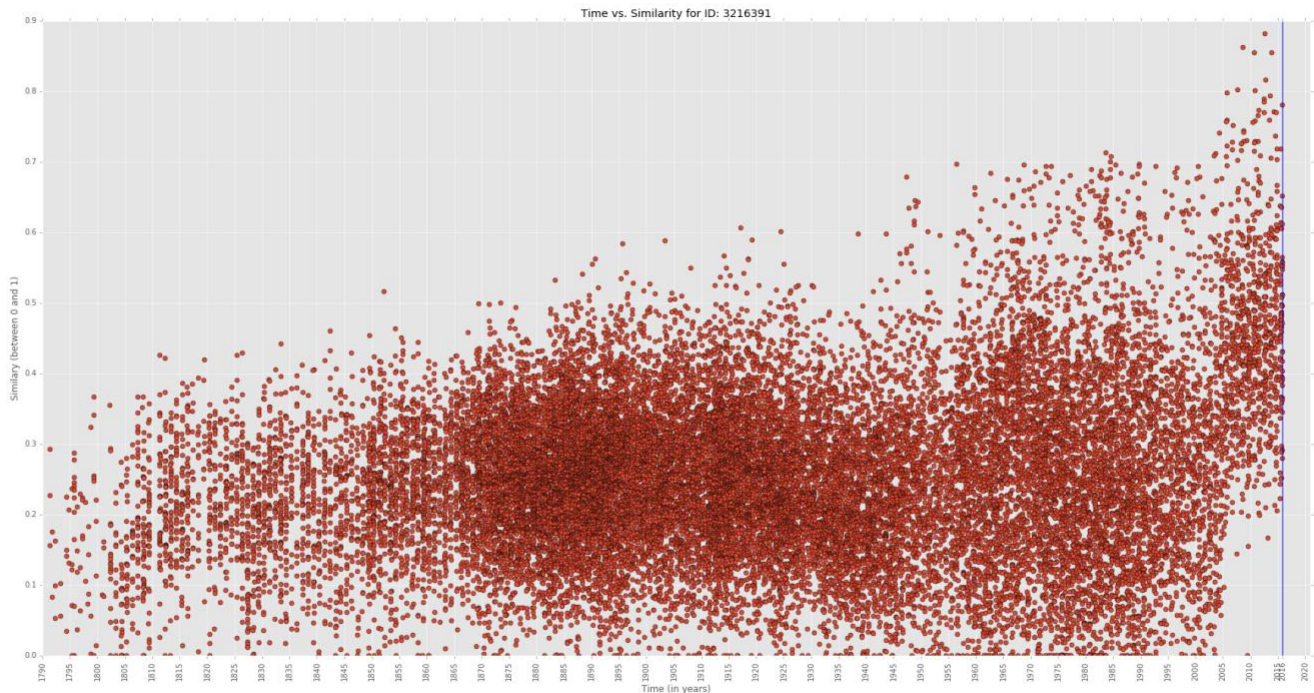
Figure 3: Similarities between each Document vs. Document 3216391 over Time

The blue vertical line is the year of the specified document, while each red dot represents a different document. We noticed that there is an odd break in the similarity around 2006 and this shows up in most post-2006 documents. As we trained this model at the end of this semester, we have yet to identify the cause. This is just a small example of how the document vectors can be used outside of clustering and classification.

9 Conclusion and Future Works

A huge first part of the report dealt with an exploratory summarization of SCOTUS Network's opinion texts using simple Network and NLP data structures, dimensionality reduction, and various clustering methods. Rather than contextualizing clusters within the limitations of the tf-idf matrix (extracting singular words), it would be worthy to look into NLP approaches that could briefly summarize a SCOTUS opinion or a set of SCOTUS opinions.

Many problems arose during this process, such as k-means for $K = 1000$ and GMM and HCC not working for the tf-idf and bow matrices. A potential fix is using supercomputing resources, such as UNC's KillDevil/Kluster Computing **[15]** to perform computations requiring huge memory, i.e. converting the large, sparse matrices into their dense forms. This would allow for doing other

dimensionality reductions that require dense representations, such as principal component analysis (PCA) because it centers the data (for each element, subtract off the mean of feature) before applying SVD **[14]**. There is also the question of the need for more or better summarize cluster functions.

Quantitative measures on how well clustering methods performed was also not accounted for, such as calculating the modularity score for each cluster or using the clustering metrics in scikit that compare the relative performance between two clustering methods **[4]**. Number of components/clusters were chosen arbitrarily for all of the clustering methods, as the luxury for running multiple initial configurations for criterion checks, such as elbow criterion for k-means or BIC criterion for GMM **[14],** was not possible due to time constraints. Overall, although there were many problems, we hope the shiny app is still of use to both statistics and law.

As for deep learning, there is great potential for word2vec and doc2vec to be used to increase an understanding of the SCOTUS cases. As a next step, we could see potentially useful results from using the document embeddings for clustering. This provides a sort of semantic understanding of the documents, the extent of which is not attainable when working with other word-frequency based data. We also see deep learning as being useful for other tasks, e.g. classification, which we are beginning to explore.

Overall, the statistically-based NLP techniques we have implemented this semester have proven useful in clustering, classifying, and grouping the SCOTUS cases. We have progressed a great deal and we see many directions this research can continue.

10   References

[1] Arnold, Taylor, and Lauren Tilton. *Humanities Data in R: Exploring Networks, Geospatial Data, Images, and Text*. **Berlin: Springer, 2015. Print.**

[2] Kolaczyk, Eric D. *Statistical Analysis of Network Data: Methods and Models*. **New York: Springer, 2010. Print.**

[3] *The Supreme Court Database*. **N.p., n.d. Web. 09 May 2017.** <**http://scdb.wustl.edu/index.php**>**.**

[4] "Scikit-learn." *Scikit-learn: Machine Learning in Python — Scikit-learn 0.18.1 Documentation*. **N.p., n.d. Web. 09 May 2017.** <**http://scikit-learn.org/stable/**>**.**

[5] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schulttze. *Introduction to Information Retrieval*. **New York: Cambridge UP, 2009. Print.**

[6] Murphy, Kevin P. *Machine Learning: A Probabilistic Perspective*. Cambridge, Mass.: MIT, 2013. Print.

[7] *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. N.p., n.d. Web. 09 May 2017. <http://mathworld.wolfram.com/>.

[8] Lee, Daniel D., and Sebastian H. Seung. "Algorithms for Non-Negative Matrix Factorization." *Advances in Neural Information Processing Systems* 13 (2001): n. pag. Web.

[9] Newman, M.E.J. "Modularity and Community Structure in Networks." *Proceedings of the National Academy of Sciences of the United States* 103.23 (2006): 8577-582. Web.

[10] "Get Started with Python-igraph." *Python-igraph*. N.p., n.d. Web. 09 May 2017. <http://igraph.org/python/doc/python-igraph.pdf>.

[11] Pons, Pascal, and Matthieu Latapy. "Computing Communities in Large Networks Using Random Walks." *Physics and Society* (2005): n. pag. Web.

[12] Rudin, Walter. *Principles of Mathematical Analysis*. McGraw-Hill Education, 2013. Print.

[13] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. New York: Springer, 2015. Print.

[14] Hastie, Trevor, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning Data Mining, Inference, and Prediction*. New York: Springer, 2016. Print.

[15] "KillDevil Cluster." *UNC Information Technology Services*. N.p., n.d. Web. 09 May 2017. <http://its.unc.edu/rc-services/killdevil-cluster/>.

[16] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep Learning. Cambridge, MA: MIT, 2016. Print.

[17] Google, "Deep Learning." Udacity. Google, n.d. Web. 09 May 2017. <https://www.udacity.com/course/deep-learning--ud730>.

**[18] "Vector Representations of Words | TensorFlow." TensorFlow. Google Inc., n.d.
Web. 09 May 2017. <https://www.tensorflow.org/tutorials/word2vec>.**

**[19] Le, Quoc, and Tomas Mikolov. "Distributed Representations of Sentences and
Documents."ArXiv 32 (n.d.): n. pag. Web. 22 May 2014.**