Usage of Semantic Similarity Measurements

Tufts University Department of Computer Science

Thomas Schaffner

21 May 2016

Abstract

Study and comparison of protein function is an important research topic in modern biology and bioinformatics. Better understanding of protein function aids in targeting medical and pharmacological research. Ontologies of functional terms organize and give structure to possible protein functions, while annotation corpuses apply functional labels to specific proteins. Many methods exist to compare protein functional annotations. These methods range from simply counting the number of overlapping functional labels to more complex methods that make use of the structure of ontologies. Specifically, we look at the Resnik semantic similarity measurement. Resnik scores make use of both the structure of an ontology and the distribution of functional labels throughout an annotation corpus. In this thesis, we see that incomplete data can lead to erroneous low Resnik values, while high Resnik values are likely to be more meaningful.

Using the Gene Ontology Consortium's ontology (GO) and annotation corpuses from UniProt Swiss-Prot and the Saccharomyces Genome Database (SGD), we analyze Resnik scores. We create matrices of Resnik scores for each species, representing the Resnik values between all pairs of proteins within a species. Using these matrices, we show that even high quality datasets such as SGD and UniProt Swiss-Prot do not completely label their proteins, leaving many proteins labeled with very general functions. We go on to discuss methods for identifying high and low Resnik values. We also show that matrix completion methods do not appear effective in predicting functional similarity between two proteins within a species.

Contents

Chapter 1

Protein Comparison

1.1 Introduction

1.1.1 Proteins

Proteins perform many distinct functions within cells, from metabolism to regulation and production of other proteins. Any given species may have tens of thousands of unique proteins, each with their own roles in the life and activity of a cell. Understanding how proteins operate and interact is vital to modern medical and biological research.

By studying specific proteins, researchers can identify mechanisms at work in cancers and infectious diseases. By finding proteins involved in ailments, researchers can better target medical research or drug development. Additionally, by leveraging the genetic similarities of different species, researchers can find proteins that fill the same niche across multiple species. While developing a drug that targets a human protein may be extremely expensive, it can be cheaper and faster to experiment with proteins in a model organism that perform a similar function.

With modern technology, biologists are able to determine protein interactions and functions at unprecedented speeds. As the size of available data grows, both the possible benefits and the complexity of interpretation grow as well. Computational techniques make use of standardized semantics for discussing proteins in order to compare proteins and even predict protein function.

1.1.2 Functional Annotations

In order to discuss the functions of proteins in large data sets and across species, researchers have established conventions for identifying and labeling protein functions. These standards are broken into two pieces: functional ontologies and annotation corpuses. Ontologies represent the set of all possible functional labels, as well as the relationships between these labels. Annotation corpuses apply specific labels to individual proteins. Several different ontologies exist, and annotation corpuses differ between datasets.

Ontologies

GO, from the Gene Ontology Consortium [1], is a commonly used protein functional ontology [2]. GO provides a structured set of labels for protein function. The labels in GO are organized as a directed acyclic graph. Each node represents a functional label, while edges between nodes represent different relationships. GO contains data for six relationships between terms: "is-a", "part-of", "regulates", "positively-regulates", "negatively-regulates", and "has-part". For this thesis, we have only examined "is-a" relationships; if a functional label l_i is a specification of another term l_j, an edge exists from l_j to l_i. Note that any label may have multiple children and multiple parents. GO is partitioned into three separate domains. Each of these domains consists of its own directed acyclic graph. The root terms of these DAGs, or domain roots, are "cellular component", "biological process", and "molecular function".

Because different child terms of a single ancestor term may differ in their specificity, the "depth" of a term in the GO graph is not necessarily meaningful. For example, the GO terms for "single-organism organelle organization" and "transposition" are both children of the GO term "single-organism cellular process", and both have a depth of 4 in the GO graph. However, within the SGD annotation corpus used in this thesis, “single-organism organelle organization” is used to label more than ten times the number of proteins that “transposition” labels (279 compared to 20). Even though both terms share the same depth in GO, and are in fact siblings, they are not equally common functional labels. GO is only able to provide explicit information on the relationships between a parent and a child. It is always true that a child is more specific than a parent, but it is not necessarily true that a node is equally specific as its siblings.

The terms specified in GO are not very useful by themselves. Protein functions are more meaningful when associated with actual proteins. Annotation corpuses contain the known associations of proteins with their functions. These associations are determined either through biological experimentation or through computational prediction.

Annotation Corpuses

Different groups may publish their own annotation corpuses, based on their own experimentation or functional prediction. Different groups may also compile others’ data. Some experiments may be more error-prone than others, and computational prediction of protein function also comes with less than complete confidence. Because of these potential errors or sources of noise, annotation corpus providers may require different levels of confidence before approving a functional annotation and adding it to the corpus. All these differences lead to many slightly different datasets, potentially focusing on different species, different diseases, or proteins with specific functions.

Annotation corpuses are also far from complete. Even if all functional annotations were one hundred percent accurate, many proteins have never been tested for function. The biological experiments to determine function take time and resources, and therefore many proteins are currently overlooked and untested. Even when proteins have labels in an annotation corpus, those labels are not always terribly informative. For 1148 of 7014 proteins in the Saccharomyces Genome Database

(SGD), the most specific functional label is a domain root of GO. For roughly one seventh of the proteins in this dataset, we can provide no function more specific than “molecular function”, “biological process”, or “cellular component”.

In this thesis, we will primarily discuss two annotation corpuses: the SGD annotation corpus mentioned above, and the Swiss-Prot annotation corpus provided by UniProt. Swiss-Prot is a multi-species annotation corpus. It contains labels for proteins that belong to various species. Swiss-Prot includes only experimentally-supported functional labels. SGD provides annotations solely for *Saccharomyces cerevisiae*. The labels in SGD are also solely experimentally-derived.

1.1.3 Data Sources

Gene Ontology

All calculations were performed using the GO release of June 13, 2015. The file used was go.obo, downloaded from geneontology.org

Annotation Corpuses

Two annotation corpuses were used in this thesis.
The UniProt Swiss-Prot annotation corpus was downloaded from uniprot.org on June 23, 2015. The SGD annotation corpus was downloaded from yeastgenome.org on December 27, 2015.

1.1.4 Functional Prediction

Predicting the functional labels of experimentally untested proteins is an important research problem. Predictions can be used to identify potentially interesting proteins for medical or biological research, among other uses. Many algorithms exist for protein functional prediction, often using additional biological data. Various methods can be used to append predicted functional labels within an annotation corpus, such as simple majority voting [3] or more complex algorithms that make use of local neighborhoods within the network [4, 5], clustering within the network [6, 7, 8], and more [9, 10, 11, 12].

1.2 Techniques

In order to compare proteins, researchers look at several different aspects of each protein depending on their methodology. Proteins can be compared based on genetic sequence, structure, or the function of a protein. As discussed above, data regarding protein interactions can also be used to compare proteins and find similarities.

BLAST [13] is a tool that compares gene DNA sequence or protein amino acid sequence. BLAST produces a score measuring the similarity between sequences without examining protein functional labels, and functional annotations typically transfer if the BLAST similarity score is exceeds some threshold.

Another common tool is Pfam, which compares the secondary structure of proteins using machine learning [14]. Pfam uses hidden Markov models to classify proteins into groups with similar secondary structures.

Other techniques for comparing the function of proteins rely on existing information about the function of the proteins being compared. The simplest methods compare the exact labels of multiple proteins. For example, Jaccard GO [15, 16, 17] compares the number of functional terms shared by both proteins to the number of proteins associated with either protein. If $L(p)$ is the set of functional labels associated with protein “p”, then:

$$JaccardGO(p_i, p_j) = \frac{|L(p_i) \cap L(p_j)|}{|L(p_i) \cup L(p_j)|}$$

Other techniques compare functional labels directly, relying on the structure of GO itself. Two methods *SimUI* and *SimPE* [18] make use of induced subgraphs of GO. They define the induced subgraph of a term $V(t)$ to be all nodes and edges present in all paths from the root to the t. For example, the highlighted nodes and edges in figure 1.1 represent the induced subgraph for GO:0044700. Note that figure

1.1 refers to a partial representation of the GO DAG.

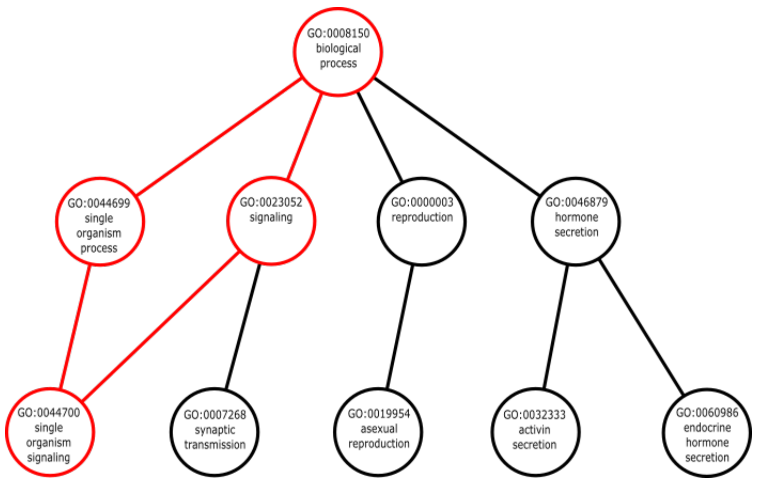

GO:0008150 biological process
GO:0044699 single organism process
GO:0023052 signaling
GO:0000003 reproduction
GO:0046879 hormone secretion
GO:0044700 single organism signaling
GO:0007268 synaptic transmission
GO:0019954 asexual reproduction
GO:0032333 activin secretion
GO:0060986 endocrine hormone secretion

Figure 1.1: Induced Subgraph of GO:0044700 on partial GO DAG

SimUI evaluates the subgraphs based on node count

$$SimUI(t_1, t_2) = \frac{|V(t_1) \cap V(t_2)|_n}{|V(t_1) \cup V(t_2)|_n}$$

while *SimPE* evaluates the subgraphs based on edge count

$$SimPE(t_1, t_2) = \frac{|V(t_1) \cap V(t_2)|_e}{|V(t_1) \cup V(t_2)|_e}$$

Some techniques even make use of the distribution of functional labels within an annotation corpus. Annotation Corpuses are often used to generate information content (IC) measurements for specific terms, based on how well-represented terms are within the annotation corpus. These information content values are used in the Resnik semantic similarity comparison, as well as all of the modifications and adaptations of the Resnik comparison. IC is also widely used in other comparisons.

1.3 Information Content Comparisons

Information content is a measurement, originally from the field of natural language processing [19], used to indicate the specificity of a given functional term in GO. Information content is calculated based on a specific annotation corpus; two IC values are not comparable unless they are both calculated using the same annotation corpus.

1.3.1 Probability

The probability of a functional label, denoted $P(l)$, is equal to the portion of proteins in the annotation corpus that perform the function of l.

$$P(l) = \frac{|proteins(l)|}{|annotation\_corpus|}$$

Note that different curators may apply labels differently; given experimental evidence that a protein p_i performs function l_i, one curator may choose to add the label l_i by itself to the annotation corpus, while another curator may also add all ancestors of l_i. Either would be technically accurate, it is a matter of convention. The structure of the GO DAG means that l_i is a specification of all of the ancestors of l_i. For example, if a protein performs a function related to synaptic transmission (GO:0007268), its function must also relate to signaling (GO:0023052).

In order to maintain consistency and accuracy when calculating probability, we define $proteins(l)$ as a function that returns all proteins in the annotation corpus that contain either l or any of the descendants of l as a functional label.

1.3.2 Information Content

If a term is close to its domain root, it is more likely to have a high probability value. Nearly all proteins in a given annotation corpus will contain the functional label "biological process" or one of biological process' descendants. Essentially, the lower the probability of a functional label, the more specific it is likely to be. Therefore, information content is defined as

$$IC(l) = -ln(P(l))$$

The least-represented GO terms in a given annotation corpus will have a low probability and consequently a higher information content value. Because the probability of a term takes into account proteins labeled with that term's descendants, the probability of a term is always at least as large as any of the probabilities of that term's children. Therefore, a term in GO always has an information content that is at least as large as its parent term's information content value.

1.4 The Resnik Score

A Resnik similarity score is an indication of the level of similarity between two functional labels within the context of a common ontology and annotation corpus [19]. Given two functional labels l_i and l_j, the maximum informative common ancestor (MICA) of those terms is the common ancestor of l_i and l_j with the highest information content.

All common ancestors of two functional labels represent functions shared by both terms. The MICA of two terms, therefore, represents the most specific or most informative shared function between the two functional terms being compared. The Resnik score of l_i and l_j is equal to the information content of the maximum informative common ancestor of l_i and l_j.

$$simRes(l_i, l_j) = IC(MICA(l_i, l_j))$$

1.4.1 Resnik Scores Across GO Domains

Resnik scores rely on the MICA of two terms. If two functional labels are from different domains of GO, then they are part of separate DAGs. Therefore, they have no common ancestors, and no MICA. In this case, GO is treated as having a true root node with three direct children: “molecular function”, “biological process”, and “cellular component” (the domain roots of GO). All functional labels in GO are descendants of this dummy root. Therefore, the probability

$$P(ROOT_{dummy}) = 1$$

The information content of the dummy root consequently equals zero. This dummy root allows us to treat the Resnik score between two functional labels from different GO domains as 0.

1.4.2 Resnik Scores and Distance

Note that Resnik scores between two GO terms are symmetric. However, a Resnik score is a measurement of similarity, not a distance metric. Resnik scores themselves do not obey the triangle inequality. For example, consider the terms “GO:0016075” (rRNA catabolic process), “GO:0007483” (genital disc morphogenesis), and “GO:0061558” (cranial ganglion maturation).

$$simRes(\text{GO:0016075}, \text{GO:0007483}) = 0.037477152772432756$$
$$simRes(\text{GO:0007483}, \text{GO:0061558}) = 3.7497179567995427$$
$$simRes(\text{GO:0016075}, \text{GO:0061558}) = 0.037477152772432756$$

This makes sense, because the Resnik comparison measures similarity rather than dissimilarity. However, even taking the multiplicative inverse of Resnik scores does not provide a true distance metric. The reciprocals of Resnik scores between the functional labels "GO:0033471" (GDP-L-galactose metabolic process), "GO:1901805" (beta-glucoside catabolic process), and "GO:1901699" (cellular response to nitrogen compound) do not obey the triangle inequality.

$$\frac{1}{simRes(\text{GO:0033471}, \text{GO:1901805})} = 0.6011489018392736$$
$$\frac{1}{simRes(\text{GO:1901805}, \text{GO:1901699})} = 26.682923488669466$$
$$\frac{1}{simRes(\text{GO:0033471}, \text{GO:1901699})} = 1.908992864684634$$

Therefore, Resnik scores are not trivially interchangeable with other distance metrics between terms.

1.4.3 Modifications to Resnik Scores

Several semantic similarity measurements tweak the Resnik comparison slightly in order to normalize or adjust sensitivity. One measurement, *simLin* [20], uses the information contents of the terms being compared to normalize Resnik scores.

$$simLin(l_i, l_j) = \frac{simRes(l_i, l_j)}{IC(l_i) + IC(l_j)}$$

1.4.4 Relative Specificity Similarity

Wu et al. have developed several other GO-based measurements of similarity between functional labels. Relative Specificity Similarity (RSS) is a similarity measurement that does not require an annotation corpus [21]. RSS relies on the most recent common ancestor (MRCA) of two terms. Wu et al. also define α, β, and γ.

For terms l_i and l_j

$$\alpha = max\left(|m \cap n|_n\right)$$

for all paths m from root to l_i and all paths n from root to l_j

$$\beta = (gen(l_i), gen(l_j))$$

where $gen(l)$ is the smallest number of edges between l and any descendant of l that is a leaf (has no descendants of its own).

$$\gamma = dist(\mathrm{MRCA}(l_i, l_j), l_i) + dist(\mathrm{MRCA}(l_i, l_j), l_j)$$

α is the relative specificity of l_i and l_j, β is the relative generality of l_i and l_j, and γ is the sum of distances between MRCA(l_i, l_j) and l_i and l_j. Based on these definitions, RSS is defined as

$$RSS(l_i, l_j) = \frac{maxDepth^{GO}}{maxDepth^{GO} + \gamma} * \frac{\alpha}{\alpha + \beta}$$

1.4.5 Hybrid Relative Specificity Similarity

Wu et al. also modified their RSS measurement to make use of information content. They created Hybrid Relative Specificity Similarity (HRSS) using both the most informative leaf (MIL) and MICA of terms, as well as a Resnik comparison [22]. They redefine α_{IC}, β_{IC}, and γ_{IC} for terms l_i and l_j as follows.

$$\alpha_{IC} = simRes(l_i, l_j)$$

$$\beta_{IC} = \frac{(IC(MIL(l_i)) - IC(l_i)) + (IC(MIL(l_j)) - IC(l_j))}{2}$$

$$\gamma_{IC} = (IC(l_i) - simRes(l_i, l_j)) + (IC(l_j) - simRes(l_i, l_j))$$

1.5 Protein Comparisons

Proteins in the annotation corpus often have more than one label, and can have multiple labels that are very dissimilar functionally. However, many of these semantic similarity measurements only compare individual terms. In order to apply these measurements to protein functional comparison, several “mixing methods” are commonly used. The simplest method for comparing two proteins p_i and p_j, and the method used in this thesis, is to simply take the maximum similarity score of all pairs of (l_i, l_j), where l_i is a functional label of p_i and l_j is a functional label of p_j. Another simple method is to take the average of all pairwise comparisons.

1.6 Discussion

1.6.1 Conditional Information Content

The relationship between the information content of a functional label and the information content of one of its descendants has a special meaning. A “conditional information content” score can indicate the similarity between an ancestor functional label and one of its descendants. Bayes’ theorem shows that the difference in information content between an ancestor term and a descendant term is based on a conditional probability, defined as follows.

$$P(l_i \mid l_j) = \frac{|proteins(l_i) \cap proteins(l_j)|}{|proteins(l_j)|}$$

With this definition, it is possible to consider conditional information content values, defined below.

$$IC(l_i|l_j) = -ln(P(l_i \mid l_j))$$

When comparing two functional labels l_i and l_j where one label is an ancestor of the other, the intersection $proteins(l_i) \cap proteins(l_j)$ is guaranteed to be non-empty as long as both $proteins(l_i)$ and $proteins(l_j)$ are both non-empty. If l_i is the ancestor of l_j, note that every protein in $proteins(l_j)$ is also in $proteins(l_i)$ by definition. Therefore, the conditional probability $P(ancestor \mid descendant)$ will always be 1. According to Bayes' theorem

$$P(desc. \mid anc.) = \frac{P(anc. \mid desc.)P(desc.)}{P(anc.)}$$

Therefore

$$\begin{aligned}
\mathrm{IC}\left(\mathrm{desc.} \mid \mathrm{anc.}\right) &= -\ln\left(\mathrm{P}\left(\mathrm{desc.} \mid \mathrm{anc.}\right)\right) \\
&= -\ln\left(\frac{\mathrm{P}\left(\mathrm{anc.} \mid \mathrm{desc.}\right)\mathrm{P}\left(\mathrm{desc.}\right)}{\mathrm{P}\left(\mathrm{anc.}\right)}\right) \\
&= -1 * \left(\ln\left(\mathrm{P}\left(\mathrm{anc.} \mid \mathrm{desc.}\right)\right) + \ln\left(\mathrm{P}\left(\mathrm{desc.}\right)\right) - \ln\left(\mathrm{anc.}\right)\right) \\
&= -\ln\left(1\right) - \ln\left(\mathrm{P}\left(\mathrm{desc.}\right)\right) + \ln\left(\mathrm{P}\left(\mathrm{anc.}\right)\right) \\
&= \mathrm{IC}\left(\mathrm{desc.}\right) - \mathrm{IC}\left(\mathrm{anc.}\right)
\end{aligned} \tag{1.1}$$

This shows that the difference between the information content values of two functional labels has a special meaning if one label is the ancestor of the other. The difference in information content values represents the relative information content of the descendant with respect to the ancestor. If the difference $IC\left(desc.\right) - IC\left(anc.\right)$ is high, then the proportion $\frac{|proteins\left(desc.\right)|}{|proteins\left(anc.\right)|}$ must be low. Therefore, this conditional information content score can be used to differentiate between functional terms that are significantly more specific than their parent, and functional terms that provide very little additional information compared to their parents.

1.6.2 Resnik Scores

Resnik scores are often discussed in relation to functional similarity problems. New measurements are often compared to the Resnik semantic similarity comparison method, which is one of the simplest information content-based measures. However, some of the issues of Resnik scores are rarely discussed.

It is important to know that Resnik scores are not true metrics. Because Resnik scores do not obey the triangle inequality (and cannot be trivially manipulated to obey the triangle inequality) they cannot be substituted for common distance metrics such as shortest path. Data quality issues are also rarely mentioned in depth. The difference in meaning and meaningfulness of high Resnik score values and low Resnik score values is often neglected. However, we have seen that many proteins in up to date annotation corpuses are only labeled with general functional terms. It is important to remember, when using Resnik scores as a comparison, that low scores do not necessarily indicate dissimilarity.

1.6.3 Resnik Scores and Data Quality

When discussing functional comparisons between proteins, the data quality issues of annotation corpuses discussed earlier impact the meaningfulness of semantic similarity measurements. In the case of Resnik scores (the primary semantic similarity measurement for the rest of this thesis), incomplete annotation corpuses lead to erroneously low estimates of the Resnik scores between some proteins.

Several situations can result in a low Resnik score. Two proteins that are truly dissimilar will always have a low Resnik score, as long as they have at least a single correct functional label. If these similar proteins are labeled with their most specific functional terms but their MICA is a domain root, they will have an extremely low Resnik score. However, incomplete labelings of similar proteins can also lead to a low Resnik score. If a protein has multiple functions but is only labeled with one function, that protein will appear dissimilar (low Resnik score) to other proteins that share the missing label as a true function. Recall that many proteins can have very general functional terms as their sole labels. Even if a function is not strictly missing, it is still possible that a protein is labeled with a more general version of its function. This can lead to lower Resnik scores, regardless of the true functional similarity of proteins.

Incomplete annotation corpuses cannot, however, lead to falsely high Resnik scores under the maximum mixing method. The Resnik score between two proteins under the maximum mixing method is equal to the highest Resnik score between any pair of terms across the proteins. Therefore, if two proteins have a high Resnik score, they must be labeled with two functionally similar terms. Added functional labels cannot reduce the Resnik score of two proteins. Only false annotations can create a falsely high Resnik score. Because the annotation corpuses used in this thesis exclusively use functional labels backed by experimental evidence, all functional labels that we used are more likely to be accurate than labels from annotation corpuses that allow computationally predicted annotations.

Chapter 2

Resnik as a Matrix

We have calculated Resnik scores for all yeast (Saccharomyces cerevisiae), mouse (Mus musculus), and human (Homo sapiens) proteins. Mouse and human Resnik scores were calculated using the UniProt Swiss-Prot annotation corpus, while yeast Resnik scores were calculated using the SGD annotation corpus.

For each species, we created a square matrix of Resnik scores. Table 2 shows the number of proteins and total Resnik scores for each species. Each row and column within a Resnik score matrix represents a single protein, with element $(i; j)$ representing the Resnik score between protein i and protein j. Because $simRes(l_i; l_j) = simRes(l_j; l_i)$, these matrices are symmetric. However, not all pairs of proteins have functional labels from the same domain of GO. Consequently, table 2 shows that a number of Resnik scores within each species are cross-domain, resulting in Resnik scores of 0.

Species	Number of Proteins	Total Entries	Cross-Domain Scores
Human	20205	408242025	65409425
Mouse	16711	279257521	19243345
yeast	7014	49196196	3876

2.1 New Properties

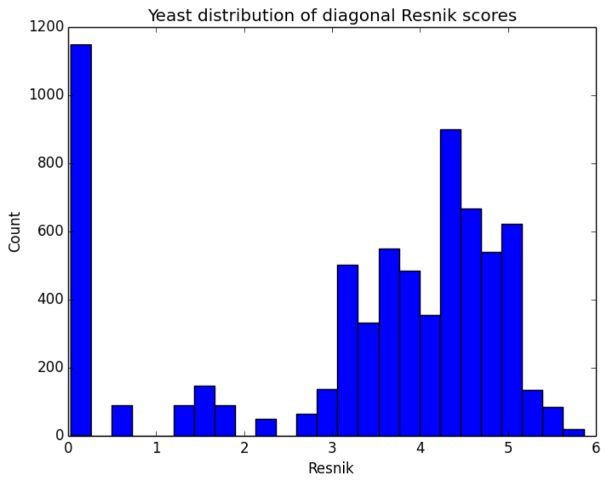
When examining a matrix of Resnik scores, it is possible to find new patterns and properties. The distribution of values can provide a basis for differentiating “high” and “low” Resnik scores. As discussed above, low Resnik scores can arise from several situations while high Resnik scores are more likely to be accurate. Distinguishing between high and low scores in the matrix can identify high-confidence Resnik comparisons. Namely, by examining the distribution of Resnik scores, it is possible to identify the high-value Resnik scores as high confidence.

2.1.1 Diagonals in Resnik Score Matrix

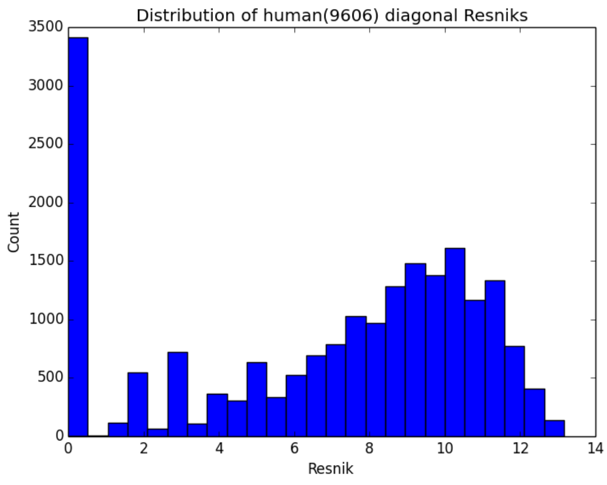
Other properties of the matrix can be used to find poorly-labeled proteins and low-confidence Resnik scores. A simple indicator of a protein’s labeling quality is the diagonal of the matrix. A diagonal at index i represents the Resnik score between $protein_i$ and itself. Under the maximum mixing method, this is equal to the information content of the most informative term of $protein_i$. Essentially, the diagonal at index i is equal to $max\left(IC(l)\right)$ for all functional labels l associated with $protein_i$. If a diagonal at index i has a low value, the information content of the most informative functional label of $protein_i$ must be low. $protein_i$ cannot, therefore, be labeled with any specific functional labels. For reference, figure 2.1 shows histograms of the diagonal values for each species’ Resnik matrix.

Diagonals as Upper Bound on Resnik Scores

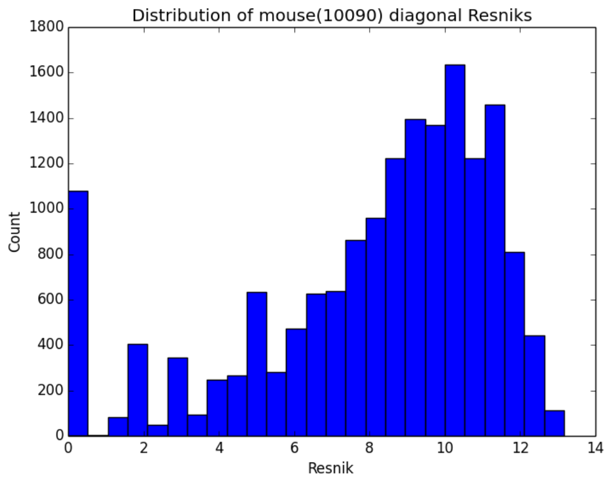
The MICA of two functional labels l_i and l_j must be an ancestor of both l_i and l_j. The information content of $MICA(l_i, l_j)$ must be less than or equal to $min\left(IC(l_i), IC(l_j)\right)$. Therefore, the Resnik similarity between two terms is bounded by the less informative term. A diagonal at index i in a Resnik matrix is an upper bound on values in column i and row i. If $protein_i$ does not have any highly informative functional labels, then $simRes(protein_i, protein_j)$ will always be low for any j. $protein_i$ can therefore be considered poorly labeled, and all comparisons with $protein_i$ (all entries



(a) SGD diagonals



(b) Human diagonals



(c) Mouse diagonals

Figure 2.1: Histograms of diagonal values in Resnik matrices

in row i or column i in the matrix) are likely not meaningful.

2.1.2 Distribution of Specificity

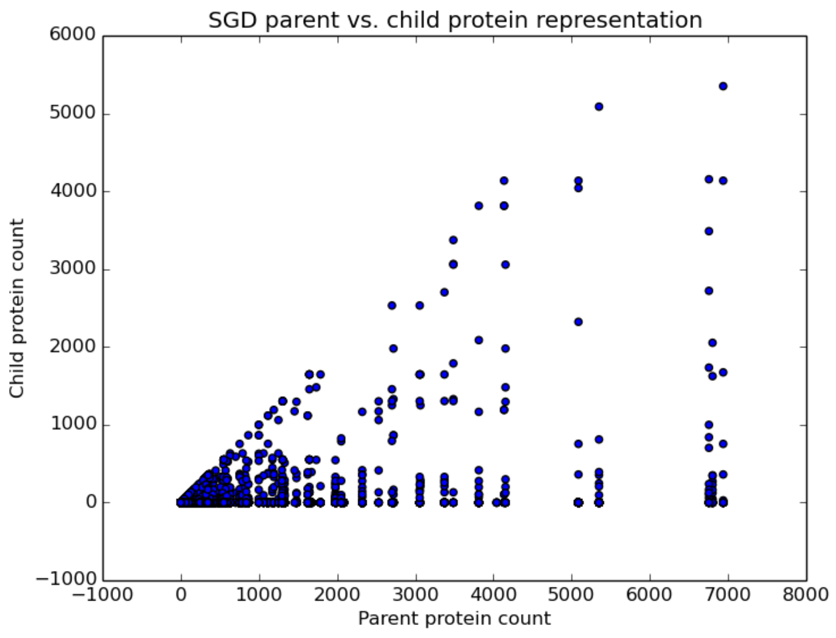
Beyond diagonals, we can use statistics to determine low or high Resnik values. In the SGD Resnik matrix (generated using the SGD annotation corpus), Resnik values greater than 3.043 are two standard deviations above the mean. In the human matrix, the cutoff is 5.4435 while in mouse the cutoff is 5.9412. Alternately, we can classify some of the lowest values as less meaningful by comparing matrix entries to the IC values of highly-represented functional labels in the annotation corpus. The graphs in figure 2.2 show the size of $proteins(label)$ and $proteins(label) \cap proteins(label_{child})$ for each child $label_{child}$ of $label$. There is one data point on the graph for each parent-child relationship within GO.

In order to make use of these graphs for preliminary testing, we chose all data points with $|proteins(parent) \cap proteins(child)| \geq 3000$ for SGD, and $|proteins(parent) \cap proteins(child)| \geq 250000$ for UniProt Swiss-Prot. This leads to IC cutoff values of roughly 0.85 and 0.74 respectively.

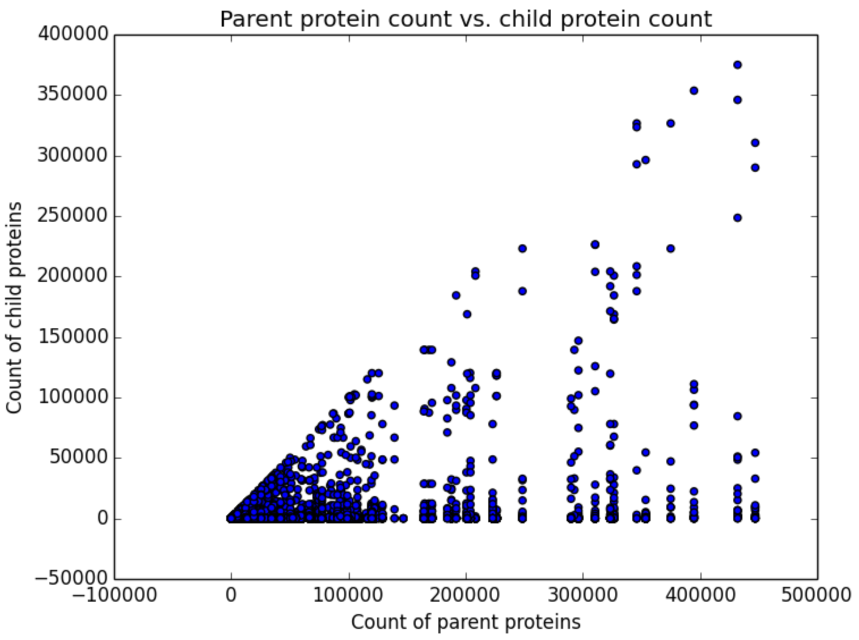
2.2 High Value Density

Looking at the SGD Resnik score matrix, we examined the density of high Resnik score values within each row of the Resnik score matrix. For this evaluation, a "high" Resnik score is a value greater than or equal to 3.043. For each row in the Resnik score matrix, we calculated the percentage of entries that were "high" values. Figure 2.3 shows a histogram with the distribution. For this histogram, we removed all percentages equal to 0 in order to make the rest of the graph more readable.

We noticed several outliers on the right of the graph. A few rows (corresponding to proteins) had exceptionally high percentages of high Resnik scores. The proteins five proteins with the highest percent of high Resnik scores were YBR160W, YER125W, YPL204W, YBR279W, and YER133W. According to yeastgenome.org,



(a) SGD annotation corpus



(b) UniProt annotation corpus

Figure 2.2: $|proteins(parent)|$ vs. $|proteins(parent) \cap proteins(child)|$

all of these proteins have regulatory functions. This similarity between the outliers may be useful for identifying regulatory proteins in the future.

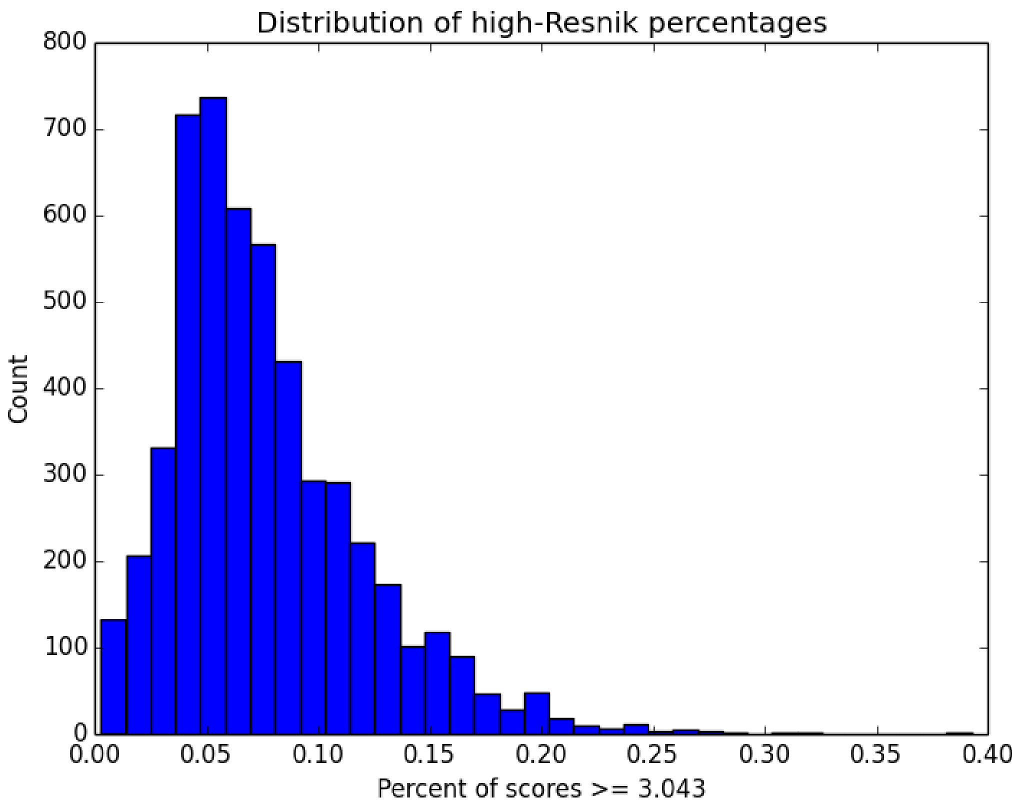


Figure 2.3: Distribution of high Resnik percentages in matrix rows

2.3 Resnik Score Cutoff

We have used a couple simple methods to classify Resnik scores as low or high in value. However, we have not performed experiments to determine the efficacy of the methods discussed in this thesis. In future work, we hope to refine and further study these methods. These methods would allow researchers to classify Resnik scores as high- or low-confidence with more accuracy and precision.

Chapter 3

Matrix Completion

3.1 Introduction

The problem of matrix completion, especially for matrices of low rank, has become widely studied recently. Matrix completion is applicable in several situations. Images can be represented as matrices, and matrix completion can be used to identify image subjects or components. In a more abstract setting, matrix completion can be used in the context of the Netflix prize; it can predict user preferences or ratings of movies.

3.2 Methods

Essentially, the matrix completion problem takes as input a matrix with some entries missing. A solution to the matrix completion problem returns a matrix with estimates in place of missing entries. We attempt to apply an existing matrix completion solution (LMaFit, translated to Python code by Professor Mark Crovella of Boston University) [23] to the Resnik matrices. LMaFit uses the nuclear norms method [24] for matrix completion. We treat low Resnik scores in the Resnik matrices as missing data entries. We then run LMaFit to estimate the missing values. Unfortunately, the preliminary results for matrix completion on Resnik matrices (discussed below in more detail) are not promising.

3.3 Results

We ran LMaFit on the SGD matrix using an IC cutoff value of 0.85 (as discussed in chapter 2). We first set all entries in the SGD matrix below 0.85 to 0. We then considered all remaining non-zero values to be "known" values, essentially high-enough confidence to keep. For each trial, we split the known values in half to run a 2-fold cross validation. Note, because the Resnik score matrices are symmetric, entries (i, j) and (j, i) were always partitioned into the same fold. For each fold, we set the opposite fold's values to zero and used LMaFit to predict non-zero values for all zeroed values in the matrix. The maximum Resnik score in the SGD matrix is roughly 5.8599, and we found an average error of 1.738 for all predicted matrix entry values using LMaFit.

For comparison, we also randomly selected entry values between 0 and the maximum value of the matrix (5.8599) to fill all zeroed entries. The average error for random completion was 1.771. Given the large error and negligible difference between LMaFit matrix completion results and random prediction results, LMaFit does not seem to accurately predict Resnik scores based on a Resnik score matrix.

3.4 Discussion

We only have preliminary results for matrix completion of Resnik score matrices. However, these results are not promising. With high error values roughly equivalent to randomly predicting missing entries, there is no evidence that matrix completion methods can predict functional similarity between proteins with any confidence. It may be worthwhile to investigate other methods for matrix completion, but there seems to be a low probability of success.

Bibliography

[1] Michael Ashburner et al. “Gene Ontology: tool for the unification of biology”. In: *Nature genetics* 25.1 (2000), pp. 25–29.

[2] Jiajie Peng et al. “Extending gene ontology with gene association networks”. In: *Bioinformatics* (2015), btv712.

[3] Benno Schwikowski, Peter Uetz, and Stanley Fields. “A network of protein–protein interactions in yeast”. In: *Nature biotechnology* 18.12 (2000), pp. 1257–1261.

[4] Hon Nian Chua, Wing-Kin Sung, and Limsoon Wong. “Exploiting indirect neighbours and topological weight to predict protein function from protein–protein interactions”. In: *Bioinformatics* 22.13 (2006), pp. 1623–1630.

[5] Haretsugu Hishigaki et al. “Assessment of prediction accuracy of protein function from protein–protein interaction data”. In: *Yeast* 18.6 (2001), pp. 523–531.

[6] Vicente Arnau, Sergio Mars, and Ignacio Marin. “Iterative cluster analysis of protein interaction data”. In: *Bioinformatics* 21.3 (2005), pp. 364–378.

[7] Gary D Bader and Christopher WV Hogue. “An automated method for finding molecular complexes in large protein interaction networks”. In: *BMC bioinformatics* 4.1 (2003), p. 1.

[8] Jimin Song and Mona Singh. “How and when should interactome-derived clusters be used to predict functional modules and protein function?” In: *Bioinformatics* 25.23 (2009), pp. 3143–3150.

[9] Elena Nabieva et al. “Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps”. In: *Bioinformatics* 21.suppl 1 (2005), pp. i302–i310.

[10] Ulas Karaoz et al. “Whole-genome annotation by using evidence integration in functional-linkage networks”. In: *Proceedings of the National Academy of Sciences of the United States of America* 101.9 (2004), pp. 2888–2893.

[11] Alexei Vazquez et al. “Global protein function prediction from protein-protein interaction networks”. In: *Nature biotechnology* 21.6 (2003), pp. 697–700.

[12] Roded Sharan, Igor Ulitsky, and Ron Shamir. “Network-based prediction of protein function”. In: *Molecular systems biology* 3.1 (2007), p. 88.

[13] Stephen F Altschul et al. “Basic local alignment search tool”. In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.

[14] Alex Bateman et al. “The Pfam protein families database”. In: *Nucleic acids research* 30.1 (2002), pp. 276–280.

[15] Paul Jaccard. *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz, 1901.

[16] Paul Jaccard. “The distribution of the flora in the alpine zone.” In: *New phytologist* 11.2 (1912), pp. 37–50.

[17] Daniele Merico et al. “Enrichment map: a network-based method for gene-set enrichment visualization and interpretation”. In: *PloS one* 5.11 (2010), e13984.

[18] Yuan-Peng Li and Bao-Liang Lu. “Semantic Similarity Definition over Gene Ontology by Further Mining of the Information Content.” In: (2008), pp. 155–164.

[19] Philip Resnik. “Using information content to evaluate semantic similarity in a taxonomy”. In: *arXiv preprint cmp-lg/9511007* (1995).

[20] Pietro H Guzzi et al. “Semantic similarity analysis of protein data: assessment with biological features and issues”. In: *Briefings in bioinformatics* 13.5 (2012), pp. 569–585.

[21] Xiaomei Wu et al. “Prediction of yeast protein–protein interaction network: insights from the Gene Ontology and annotations”. In: *Nucleic acids research* 34.7 (2006), pp. 2137–2150.

[22] Xiaomei Wu et al. “Improving the measurement of semantic similarity between gene ontology terms and gene products: insights from an edge-and IC-based hybrid method”. In: *PloS one* 8.5 (2013), e66745.

[23] Zaiwen Wen, Wotao Yin, and Yin Zhang. “Solving a low-rank factorization model for matrix completion by a nonlinear successive over-relaxation algorithm”. In: *Mathematical Programming Computation* 4.4 (2012), pp. 333–361.

[24] Emmanuel J Candès and Benjamin Recht. “Exact matrix completion via convex optimization”. In: *Foundations of Computational mathematics* 9.6 (2009), pp. 717–772.

Appendix A

SemSimCalculator

A.1 semsimcalc.py

I have produced a python class that can be used to calculate several semantic similarity measurements. The code is included as Appendix A, and the full README can be found on github, under TuftsBCB/semsimcalc. Given a file representing GO and an annotation corpus, the SemSimCalculator class can calculate probability and information content values, Resnik scores, conditional probabilities, conditional information content scores, and a couple other semantic similarity measurements. The SemSimCalculator class also contains two mixing methods for protein comparison: maximum and average. The relationship between proteins and GO terms is stored internally, so there is no need to look up labels of proteins separately.

A.2 Code

```python
#!/usr/bin/python

# See http://bib.oxfordjournals.org/content/13/5/569.full
# For definitions

import sys
import time
import networkx as nx
import math
import pickle
import numpy

# Helper functions
def announce(message):
        """  Timestamped output to stdout """
        print time.strftime('%H:%M'),message
        sys.stdout.flush()

def open_or_abort(filename, option='r'):
        """ Output error message to stderr if file opening failed """
        try:
                newfile = open(filename, option)
        except IOError:
                sys.stderr.write("Could not open {} -- Aborting\n".format(filename))
                raise IOError
        return newfile


# NOTE(tfs): Accepted GO file format:
#
#        ! comments
#
```

```
#         [Term]
#         id: GO_term
#         ...
#         is_a: GO_term
#         is_a: GO_term
#
#         [Term]
#         ...
#
#          [Typedef]
#
# The [Typedef] tag signals end of GO terms.
# It is necessary in the current implementation
#
def parse_go_file(go_file_name):
        """ Parses and returns (does not natively store) GO data """

        go_file = open_or_abort(go_file_name)

        # Setup
        go_file.seek(0)
        go_graph = nx.DiGraph()

        alt_ids = {}

        go_term = ''
        parents = []
        is_obsolete = False

        # Don't start paying attention until we see '[Term]'
        valid_to_read = False

        # Main parsing loop
        for line in go_file:

                # Only if we're within a '[Term]' header
                if valid_to_read:
                        if line.startswith('alt_id:'):
                                alt_id = line.strip()[8:]
                                alt_ids[alt_id] = go_term

                        elif line.startswith('id:'):

                                # Only log if the entry is valid
                                if not is_obsolete:
                                        if go_term != '':

                                                # Only add connected node
                                                if len(parents) > 0:
                                                        go_graph.add_node(go_term)

                                                        for parent in parents:
                                                                if parent != '':
                                                                        go_graph.add_edge(parent, go_term)

                                # Reset regardless of logging status
                                parents = []
                                is_obsolete = False

                                go_term = line.strip()[4:]
                        elif line.startswith('is_a:'):

                                # Store is_a as a parent
                                parents.append(line.split('!')[0].strip()[6:])

                        elif line.startswith('is_obsolete: true'):

                                # Do not store the data under this '[Term]' header
                                is_obsolete = True

                        elif line.startswith('[Typedef]'):
                                # Write if the previous entries were valid

                                # Only log if the entry is valid
                                if not is_obsolete:
                                        if go_term != '':

                                                # Only add connected node
                                                if len(parents) > 0:

                                                        go_graph.add_node(go_term)

                                                        for parent in parents:
                                                                if parent != '':
                                                                        go_graph.add_edge(parent, go_term)

                                # Reset regardless of logging status
                                parents = []
                                is_obsolete = False

                                go_term = '' # No valid ID to reset with under a '[Typedef]' header

                                # Stop paying attention
                                valid_to_read = False
                else:
                        if '[Term]' in line:

                                # Start paying attention
                                valid_to_read = True

        go_file.close()

        return (go_graph, alt_ids)


# NOTE(tfs): Accepted AC file format:
#
#         -
#        protein_name
#        GO_term
#        GO_term
#        GO_term
#         -
#
def parse_annotation_corpus(ac_file_name, alt_ids=None):
        """
                Parses annotation corpus. Returns a dictionary of { gene: [terms] }.
                If a term is a key in alt_ids, saves the associated value instead (if provided).
        """

        ac_file = open_or_abort(ac_file_name)

        # Setup
        prot_to_gos = {}
        go_to_prots = {}

        ac_file.seek(0)

        curr_prot = ''
        curr_gos = []
        new_entry = True

        for line in ac_file:

                # Start information from new entry
                if line.startswith('-'):

                        # Only update if we have enough information for the last entry
                        if curr_prot != '' and len(curr_gos) > 0:

                                # Update prot_to_gos
                                if curr_prot in prot_to_gos:
                                        prot_to_gos[curr_prot] = prot_to_gos[curr_prot] + curr_gos
                                else:
                                        prot_to_gos[curr_prot] = curr_gos

                # Update go_to_prots
                for go in curr_gos:
                    if go in go_to_prots:
                        go_to_prots[go].append(curr_prot)
                    else:
                        go_to_prots[go] = [curr_prot]

            # Reset, regardless of whether or not we updated
            curr_prot = ''
            curr_gos = []
            new_entry = True

        # If we've just started looking at a new entry, parse as protein name
        # DON'T do this if we're still on the delimiter line ('-')
        elif new_entry:
            curr_prot = line.strip().strip(';')
            new_entry = False
        # Otherwise, parse as GO term
        else:
            if ("GO:" in line):
                new_go = line.strip().strip(';')
                if alt_ids is not None:
                    if new_go in alt_ids:
                        new_go = alt_ids[new_go]
                curr_gos.append(line.strip().strip(';'))

    ac_file.close()

    return (prot_to_gos, go_to_prots)


# Load a saved SemSimCalculator
def load_semsimcalc(saved_path):
    """
        Loads (unpickles) a saved SemSimCalculator
    """

    return pickle.load(open(saved_path, 'rb'))





###############################
### SemSim_Calculator class ###
###############################


class SemSimCalculator():
    """
        Stores GO and annotation corpus data internally.
        Calculates different semantic similarity metrics.
    """

    def __init__(self, go_file_name, ac_file_name):
        """ Initialize using GO and annotation corpus files (pass in file name, not file object) """

        self._go_graph, self._alt_list = parse_go_file(go_file_name)
        self._prot_to_gos, self._go_to_prots = parse_annotation_corpus(ac_file_name, self._alt_list)
        self._proteins = [x[0] for x in self._prot_to_gos.items()]
        self._num_proteins = len(self._proteins)
        self._ic_vals = {} # For memoizing IC values (they are unchanging given an ontology and annotation corpus

        self._go_terms = self._go_graph.nodes()

        self._mica_store = None

    def link_mica_store(self, mica_store):
        """ Stores a reference to a MicaStore instance """

        self._mica_store = mica_store

    def unlink_mica_store(self):
        """ Removes link to a MicaStore instance (sets to None) """

        self._mica_store = None

    def save(self, filepath):
        """
            Saves (pickles) to filepath

            NOTE: Does not save reference to MicaStore instance (as this will likely be broken on load)
        """

        # Do not store reference to MicaStore instance
        temp = self._mica_store
        self._mica_store = None

        pickle.dump(self, open(filepath, 'wb'))

        # Restore _mica_store reference
        self._mica_store = temp

    def get_go_graph(self):
        """ Return nx graph for GO """

        return nx.DiGraph(self._go_graph)

    def get_alt_list(self):
        """ Return alt_list """

        return dict(self._alt_list)

    def get_ptg(self):
        """ Return copy of prot_to_gos """

        return dict(self._prot_to_gos)

    def get_gtp(self):
        """ Return copy of go_to_prots """

        return dict(self._go_to_prots)

    def get_proteins(self):
        """ Return copy of proteins """

        return list(self._proteins)

    def get_num_proteins(self):
        """ Return number of proteins """

        return int(self._num_proteins)

    def get_ic_vals(self):
        """
            Return all stored ic_vals.
            Not all values are guaranteed to exist.
            Consider running precompute_ic_vals first.
        """

        return dict(self._ic_vals)

    def get_go_terms(self):
        """ Return list of GO terms """

        return list(self._go_terms)

    def get_mica_store(self):
        """ Returns copy of mica_store """

        return self._mica_store

    def calc_term_prob(self, term):
        """ Probability of term or desc(term) to occur as a label within the annotation corpus """

        if term == None or (not term in self._go_graph):
```

```
                                return None

                        # Find all descendants of term, including term
                        terms = nx.algorithms.dag.descendants(self._go_graph, term)
                        terms.add(term)

                        annotated_proteins = {}

                        # Mark any protein labeled with term or a descendant of term
                        for term in terms:
                                if term in self._go_to_prots:
                                        for prot in self._go_to_prots[term]:
                                                annotated_proteins[prot] = True

                        prob = float(len(annotated_proteins.items())) / float(self._num_proteins)

                        return prob

        def calc_conditional_prob(self, term, condition):
                """
                        Probability that term or desc(term) appears
                        as label in annotaiton corpus,
                        given that condition appears as a term.
                """

                if term == None or (not term in self._go_graph):
                        return None

                # Find all descendants of condition, including condition
                cond_terms = nx.algorithms.dag.descendants(self._go_graph, condition)
                cond_terms.add(condition)

                # Find all descendants of term, including term
                terms = nx.algorithms.dag.descendants(self._go_graph, term)
                terms.add(term)

                conditional_proteins = {}
                for cond_term in cond_terms:
                        if cond_term in self._go_to_prots:
                                for prot in self._go_to_prots[cond_term]:
                                        conditional_proteins[prot] = True

                restricted_term_proteins = {}
                for r_term in terms:
                        if r_term in self._go_to_prots:
                                for prot in self._go_to_prots[r_term]:
                                        if prot in conditional_proteins.keys():
                                                restricted_term_proteins[prot] = True

                if len(conditional_proteins.items()) == 0:
                        return None
                else:
                        prob = float(len(restricted_term_proteins.items()))
                        prob = prob / float(len(conditional_proteins.items()))
                        return prob

        def IC(self, term):
                """ Information content: IC(c) = -log(p(c)) """

                # Check if IC has been computed for term already
                if not (term in self._ic_vals):
                        # If not seen before, compute IC
                        prob = self.calc_term_prob(term)

                        if prob == 0 or prob == None:
                                self._ic_vals[term] = None
                                return None
                        else:
                                ic = (-1) * math.log(prob)
                                self._ic_vals[term] = ic # Memoize IC value
                                return ic
                else:
                        # If seen before, return memoized value
```

```
                                return self._ic_vals[term]

        def conditional_IC(self, term, condition):
                """ Conditional Information Content: cIC(t | c) = -log(p(t | c)) """

                # Too many values to memoize
                cond_prob = self.calc_conditional_prob(term, condition)

                if cond_prob == 0 or cond_prob == None:
                        return None
                else:
                        cic = (-1) * math.log(cond_prob)
                        return cic

        def precompute_ic_vals(self):
                """ Compute and store IC values for all ontology terms """

                for term in self._go_graph.nodes():
                        self.IC(term)

        def MICA(self, left, right):
                """
                        Maximum Informative Common Ancestor:
                        MICA(t1, t2) = arg max, IC(tj)
                                                        tj in ancestors(t1, t2)

                        (returns a term, common ancestor of left and right)

                        NOTE: If a MicaStore instance is linked, first try querying the stored instance
                """

                if not left in self._go_terms:
                        if left in self._alt_list:
                                left = self._alt_list[left]
                        else:
                                return None

                if not right in self._go_terms:
                        if right in self._alt_list:
                                right = self._alt_list[right]
                        else:
                                return None

                # Attempt lookup in linked MicaStore instance
                if (self._mica_store != None):
                        mica = self._mica_store.mica_lookup(left, right)

                        if (mica != None) and (mica != ’’) and (mica != ’None’):
                                return mica
                                #if (mica == ’’):
                                        # MICA is stored, but does not exist (None is a possible MICA value)
                                #        return None
                                #else:
                                #        return mica

                # Fall through and calculate MICA

                # Find common ancestors as intersection of two ancestor sets
                # NOTE(tfs): Python sets are very slow. List comprehensions are faster
                left_ancs = nx.algorithms.dag.ancestors(self._go_graph, left)
                left_ancs.add(left)
                right_ancs = nx.algorithms.dag.ancestors(self._go_graph, right)
                right_ancs.add(right)
                ancestors = [a for a in left_ancs if a in right_ancs]

                # Edge case where left and right are the same. Treat left and right as a common ancestor
                #if left == right:
                #        ancestors.append(left)

                max_term = None
                max_IC = 0

                # Calculate IC for all ancestors; store maximum IC value and term
```

```python
		for ancestor in ancestors:
			anc_IC = self.IC(ancestor)
			if anc_IC != None and anc_IC > max_IC:
				max_IC = anc_IC
				max_term = ancestor

		return max_term

	def simRes(self, left, right):
		"""
			simRes(t1, t2) = IC[MICA(t1, t2)]
			Returns a value (IC result)
		"""

		return self.IC(self.MICA(left, right))

	def simLin(self, left, right):
		"""
			simLin(t1, t2) = [IC[MICA(t1, t2)]] / [IC(t1) + IC(t2)]
			Returns a value
			Currently untested
		"""

		leftIC = self.IC(left)
		rightIC = self.IC(right)

		if leftIC == None or rightIC == None:
			return None
		else:
			return self.IC(self.MICA(left, right)) / (leftIC + rightIC)


	def simJC(self, left, right):
		"""
			simJC(t1, t2) = 1 - IC(t1) + IC(t2) - 2xIC[MICA(t1, t2)]
			Returns a value
			Currently untested
		"""

		leftIC = self.IC(left)
		rightIC = self.IC(right)

		if leftIC == None or rightIC == None:
			return None
		else:
			return 1 - self.IC(left) + self.IC(right) - (2*self.IC(self.MICA(left, right)))

	def pairwise_average_term_comp(self, lefts, rights, metric):
		"""
			Compares each pair of terms in two sets or lists of terms.
			Returns the average of these comparison scores.
			Uses metric(left, right) to make each comparison.
			metric must take in two ontology terms (left and right) and return a numeric score.
		"""

		total_score = 0
		num_scores = 0
		for left in lefts:
			for right in rights:
				new_score = metric(left, right)

				# Count a new_score of None in the denominator, but treat it as a value of 0
				# This mimics a dummy root node if there are multiple roots in the GO DiGraph
				if new_score != None:
					total_score += new_score
				num_scores += 1

		if total_score == 0:
			return None
		else:
			return total_score / num_scores

	def pairwise_max_term_comp(self, lefts, rights, metric):
		"""
			Compares each pair of terms in two sets or lists of terms.
			Returns the maximum score found in these comparisons.
			Uses metric(left, right) to make each comparison.
			metric must take in two ontology terms (left and right) and return a numeric score.
		"""

		if (len(lefts) == 0) or (len(rights) == 0):
			return None

		max_score = 0

		for left in lefts:
			for right in rights:
				temp_score = metric(left, right)
				if temp_score != None and temp_score > max_score:
					max_score = temp_score

		return max_score

	def average_protein_comp(self, left_prot, right_prot, metric):
		"""
			Looks up all go terms for left_prot and right_prot.
			Uses pairwise_average_term_comp to compare the above sets of terms.
			metric must take in two ontology terms (left and right) and return a numeric score.
		"""

		left_terms = self._prot_to_gos[left_prot]
		right_terms = self._prot_to_gos[right_prot]

		return self.pairwise_average_term_comp(left_terms, right_terms, metric)

	def max_protein_comp(self, left_prot, right_prot, metric):
		"""
			Looks up all terms for left_prot and right_prot.
			Uses pairwise_max_term_comp to compare the above sets of terms.
			metric must take in two ontology terms (left and right) and return a numeric score.
		"""

		left_terms = []
		right_terms = []

		if (left_prot in self._prot_to_gos):
			left_terms = self._prot_to_gos[left_prot]

		if (right_prot in self._prot_to_gos):
			right_terms = self._prot_to_gos[right_prot]

		return self.pairwise_max_term_comp(left_terms, right_terms, metric)



####################################
###  End SemSim_Calculator class ###
####################################





#######################
### MicaStore class ###
#######################


class MicaStore():
	"""
		Loads a matrix of MICA scores (and a list of GO term indices),
		Provides accessors for MICA score lookup
	"""

	def __init__(self, matrix_filename, ordering_filename):
		"""
			Loads the .npy numpy array,  matrix_filename,
			Stores the indices for each GO term in ordering_filename
		"""
		orderfile = open_or_abort(ordering_filename)

		self._micas = numpy.load(matrix_filename)
		self._go_to_index = {}

		index = 0
		for line in orderfile:
			self._go_to_index[line.strip()] = index
			index += 1

		orderfile.close()

	def get_micas(self):
		"""
			Returns reference to numpy matrix of MICA values.
			NOTE: This is a large matrix
		"""

		return self._micas

	def get_ordering(self):
		"""
			Returns copy of the dictionary mapping
			GO terms to indices in the _micas matrix
		"""

		return dict(self._go_to_index)

	def get_index(self, term):
		"""
			Returns the index of a GO term in the ordering of _micas (using _go_to_index)
			Returns None if term is not in _go_to_index
		"""

		if (term in self._go_to_index):
			return self._go_to_index[term]
		else:
			return None

	def mica_lookup(self, left, right):
		"""
			If a MICA value can be found in _micas, return that MICA
			Else, return None
		"""

		left_index = self.get_index(left)
		right_index = self.get_index(right)

		if (left_index != None) and (right_index != None):
			mica = self._micas[left_index, right_index]
		else:
			mica = None

		if (mica == ''):
			# Indicates that the mica was found, but does not exist (None is a valid MICA value)
			mica = ''

		return mica


###########################
### End MicaStore class ###
###########################
```