Research in Industrial Projects for Students

Sponsor

Praedicat, Inc.

Technical Report

Information Extraction and Aggregation from Unstructured Web Data for Business Profiling

Student Members

Liang Shi, *New York University*
ls4081@nyu.edu

Alexander Michels, *Westminster College*
michac22@wclive.westminster.edu

Himanshu Ahuja, *Delhi College of Engineering*
himanshuahuja_bt2k15@dtu.ac.in

Academic Mentor

Shadi Shahsavary
shadi.shahsavary@gmail.com

Sponsoring Mentors

Dr. Stephen DeSalvo
stephen.desalvo@praedicat.com

Date: August 24th, 2018

This project was jointly supported by Praedicat, Inc. and NSF Grant DMS-0931852 .

Abstract

Analysts at Praedicat, Inc., need to manually profile companies for modeling actuarial risks. With a plethora of companies and business activities, manual search is a tedious process. Further, the analysis is generally performed on unstructured, non-uniform, and sporadic websites which makes it difficult to algorithmically search for the information needed and complex to determine the semantic meaning of the documents even when they are found. Our work attempts to tackle these problems by building an information extraction, classification, and aggregation system which procures information and compares the statements found in the documents to a credible knowledge base. Based on computational fact-checking, we are hoping this approach will produce a better information extraction and aggregation system.

Acknowledgments

The team would like to thank the Institute for Pure and Applied Mathematics, Praedicat, Inc., and the National Science Foundation for giving us this amazing opportunity. In particular, our work would not have been possible without the close guidance of Shadi Shasavari, Dr. Stephen DeSalvo, and Urjit Patel.

We cannot thank the Institute for Pure and Applied Mathematics enough for giving us an opportunity to do what we love in an interesting way on a real-world problem. We are grateful for the entire staff at IPAM for everything they did to make us feel at home. In particular, the meticulous administrative support provided by Susana Serna, efficient communication provided by Dimi Mavalski and patient technical support provided by David Medina have all led us to a smooth completion of this project.

We are also immensely grateful for the entire team at Praedicat for their hospitality, encouragement, advice, and help during the project. In addition to Stephen and Urjit, we would like to specifically thank the analyst team at Praedicat, Sheryll, Riley, and Chris, for working so closely with us to help us understand their process. Thanks are also due to Melissa Boudreau who gave us insights into the world of actuarial sciences and Dr. Robert T. Reville for graciously giving us his time and wisdom.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Praedicat, Inc, an InsurTech company in Los Angeles, CA, uses forward-looking models to predict mass litigation events for which little to no historical data exist. They offer litigation risk analytics products to insurers and companies that are concerned about their exposure. These products identify and quantify emerging risks in the property and casualty insurance market. To approach the problem of modeling risks posed by substances which have the potential to cause harm in humans, Praedicat mines scientific literature to identify risks and forecast the evolution of the scientific literature. After identifying which substances are likely to trigger mass litigation events, Praedicat is able to use this information in their proprietary models to assign a dollar value to the risk associated with each company's use of potentially hazardous substance.

Currently, a team of analysts manually tracks down and identifies which companies use the hazardous substances. This is a tedious and time-consuming process which requires them to send queries to a search engine, scan the results for pages they think may be relevant, and ultimately try to identify information which can be used as evidence to tie a company to a business practice that could potentially bring litigation. With tens of thousands of companies to be analyzed, analysts are looking for a tool to extract, classify, and aggregate relevant information so they can spend less time navigating websites and more time assessing the information.

Problem statement How to automate information extraction, classification and aggregation from unstructured data on the Internet for business profiling

Our approach We used a variety of techniques from the fields of Information Extraction and Computational Fact-Checking to acquire relevant and high integrity data. Our approach can be thought of as a positive feedback loop that discovers new facts, classifies the relevance of the statements, verifies the credibility of the fact, adds credible facts to the system, and uses these new facts to direct the system's search.

Overview We begin in Chapter 2 with a Literature Review which summarizes the work in Information Extraction and Computational Fact-Checking that we have built on. Chapter 3 describes our contributions and in Chapter 4 we discuss the results of our work. We conclude and discuss future work in Chapter 5. Additionally, Appendix A serves as a Glossary of terms and Appendix B is a list of Abbreviations. For readers who are new to Graph Theory, Appendix C contains a brief introduction. For technical details on our work, the documentation for our code is in Appendix D.

Chapter 2

Literature Review

> As far as we are capable of knowledge, we sin in neglecting to acquire it.
>
> Gottfried Wilhelm Leibniz

2.1 Information Extraction

While there is a staggering amount of information available in the world, much of it is off-limits for computers. This is because the data is unstructured, meaning there is no uniform organizational scheme or pre-defined model for the information. This is the kind of information we are used to: facts simply haphazardly strewn across book pages and websites.

Adding to the difficulty is that much of this information has been encoded in an encryption scheme so elaborate that so far no machine has truly been able to crack it: natural human language. Our languages are not only dynamic and complex, they generally don't have set rules to follow and there is no idea of a "semantic unit" within natural human language: a unit of our speech which by itself conveys semantic meaning which is independent of the context. We generally think of words as semantic units, giving words like *"kick"* definitions which they should follow, but then we construct phrases and idioms like *"kick the bucket"* that break the mold.

Information extraction is the task of automatically converting unstructured or semi-structured machine-readable documents into structured information as shown in Figure 2.1. Traditionally, it has relied on hand-crafted rules and examples and a variety of Natural Language Processing techniques such as Named-Entity Recognition and dependency parsers, but as these methods do not scale well and are not as effective on heterogeneous corpora [1]. In 2007, Banko et al. introduced the world to an interesting new concept called **Open Information Extraction**, which no longer required human intervention or such computationally taxing techniques and was capable of automatically discovering and extracting relations of interest [1]. This makes them more efficient, more robust, and allows them to extract an unlimited number of relations.

Open Information Extraction systems generally extract information in the form of subject-predicate-object triples. These are tuples in which the subject and predicate are concepts or entities such as nouns and the predicate describes a relationship between the

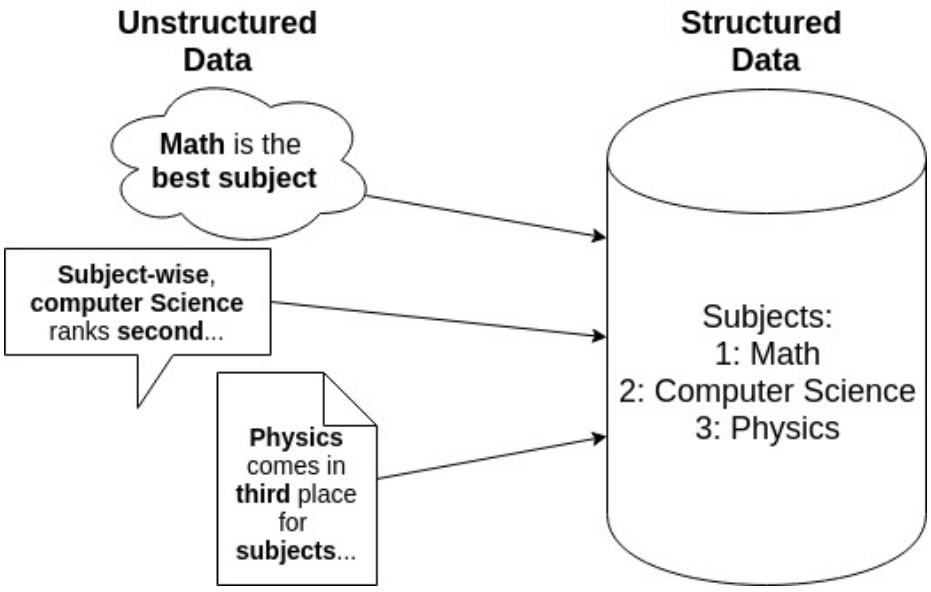


Figure 2.1: An example of how Information Extraction takes Unstructured Data and converts it to Structured Data.

the subject and object. As an example, given the sentence *"Mr. X was born on date Y"*, the system might extract the subject-predicate-object triple (*"Mr. X", "born on", "date Y"*) which could be used to fill Mr. X's date of birth field in a database.

Although Open Information Extraction systems are quite powerful, they have their limitations. One of their most significant problems is the tendency to make incoherent extractions. Incoherent extractions are relational phrases which have no meaningful interpretations. They occur because the extractor decides whether to include each word in a relational phrase on a case by case basis which often results in incomprehensible phrases such as "was central torpedo". Incoherent extractions account for 1-7% of Open IE system outputs. Approaches to solving this problem revolve around syntactic constraints on removal and inclusion of words. Fader et al. propose that every multi-word relation phrase must begin with a verb, end with a preposition, and be contiguous sequence of words in the sentence and the Reverb[1] architecture tackles the problem by imposing similar syntactic constraints [3, 4].

Another crucial problem with Open Information Extraction is their production of uninformative extractions. Uninformative extractions are those that omit critical information from the text which results in a tuple that no longer has any useful information and account for below 4-7% of Open IE system outputs. Consider the simple sentence, *"Faust made a deal with the devil"*. This approach would yield ("Faust", "made", "a deal") or ("Faust", "made", "the devil") both of which are wrong in this case because this sentence contains a light verb construction which is a multi-word expression composed of a verb and a noun where the noun carries the semantic content of the predicate (i.e. "making the deal") [3]. English has a variety of these problems resulting in extractions which have no valid semantic interpretations.

When attempting to use information obtained from Open Information Extraction, there is another big hurdle. Ironically, this is a problem with many names: Entity Resolution, Record Linkage, Object Identification, Instance Matching, and Deduplication, but it refers

[1] `http://reverb.cs.washington.edu/`

to the problem of knowing when two different labels are referring to same thing [10]. For example, "Barack Obama" and "Obama", but not all are so easy. A closely related problem is relation resolution, knowing when two relations have the same semantic meaning [10]. For example, ("X", "was founded by", "Y") and ("X", "was originally founded by", "Y") mean the same thing, but some small changes like that can drastically alter the meaning of a relational phrase, so it can be difficult to know when to merge two tuples.

2.1.1 Classification of Information

These and many other complications that are common in Information Extraction lead to a need for classification techniques for data. The abundance of data on the Internet, the low credibility of many web sources, the structural variety of its representation and presentation, and the amount of "filler" mean that systems will generally extract lots of information that is useless. The problem then becomes: given a set of documents $\mathbf{D}$, with each document d_i containing a set of statements $\mathbf{S}_i$, how does one distinguish which statements are worth including in their structured data set?

Term Frequency-Inverse Document Frequency (TF-IDF)

One of the simplest and most commonly used scoring mechanisms in Information Extraction is **Term Frequency-Inverse Document Frequency** or **TF-IDF**. The Term Frequency part of TF-IDF refers to the number of times a keyword appears in a document and we measure the magnitude of this using $log(1 + f_{t,d})$ for term t in a document d. The problem with term frequency alone is that it doesn't take the context of the set of documents into account and quantity can be misleading when it comes to keyword frequency. For example, the most frequent words ("the", "a", "is"...) won't provide any meaningful information while extremely rare words are likely to be typos or otherwise won't have a large effect on the meaning of the document because of how sparse they are.

To account for this, we also use the Inverse Document Frequency in our statistic. Inverse Document Frequency refers to the the inverse of the proportion of documents a term appears in. With this we are able to penalize words that appear in most of the documents such as "the" and "a". When the two parts are multiplied together, they yield TF-IDF. Thus the TF-IDF for a term t in a document $d_i \in \mathbf{D}$ is

$$\textbf{TF-IDF} = log(1 + f_{t,d}) \cdot log\Big(\frac{|\mathbf{D}|}{df_t}\Big) \tag{2.1}$$

where df_t is the number of documents the term appears in (document frequency) and its overall effect can be visualized in Figure 2.2 [9].

Word2Vec

One of the biggest tasks in computing is knowledge representation: how do we represent data in a way that is optimal for a computer to utilize it? This fundamental question is generally overlooked, but can make or break an application and is essential to Natural Language Processing. Natural languages send information as sequences of words, but we are trying to convey emotions, thoughts, and other incredibly complex concepts through them. We aren't trying to express to others a series of words, but instead we are trying to express a complex idea by piecing together the semantic meanings that the words represent, hoping the linguistic combination will invoke the same ideas in the other person. In our

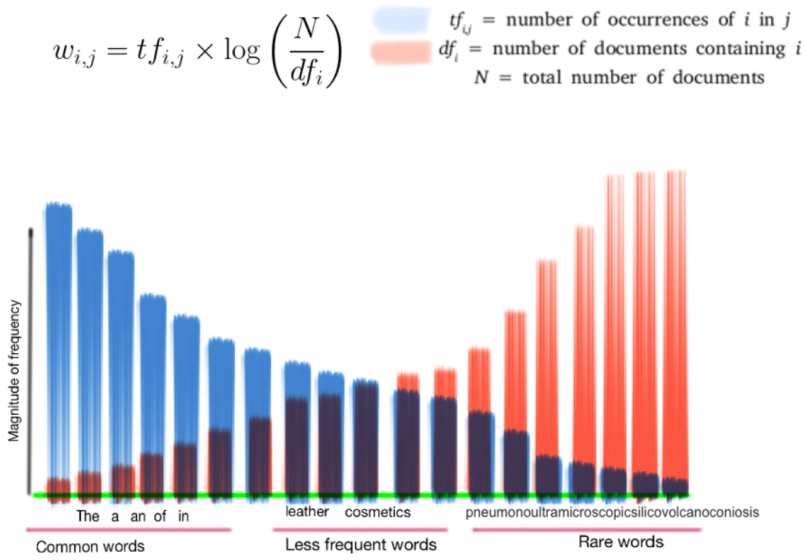


Figure 2.2: An example of how Term Frequency-Inverse Document Frequency works.

case, we are trying to capture a relation or fact and words are less than ideal for capturing this.

This makes Natural Language Processing incredibly difficult: how do we teach a computer the semantic meaning of a sequence of characters? A semantic representation means that we can no longer think about words as atomic units because we need to be able to compare the similarity of the semantic meanings of words [13]. It should also be possible to look at multiple degrees of similarity. Another useful characteristic would be the ability to do simple algebraic operations such as addition and subtraction on the representations, to be able to extract and add or subtract the *"maleness"* or *"royalty"* of a word as shown in Figure 2.3 [11]. One way we can attempt to do this is by representing words in a vector space with Word2Vec.

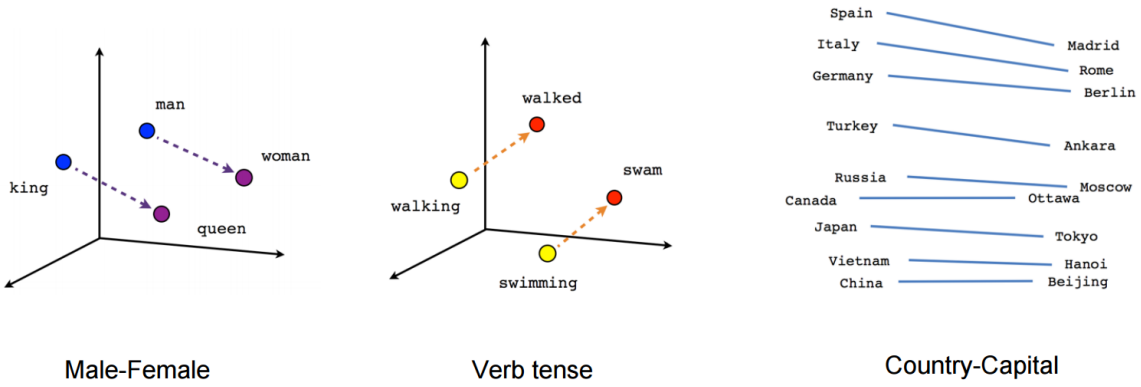


Figure 2.3: Examples of how the compositionality of word vectors works in Word2Vec [12]

.

Using vectors give us all of the desirable qualities we want in a semantic representation of a word. As long as the vectors are all created with the same dimensionality or number of features, they all exist in the same space making them easily comparable with cosine

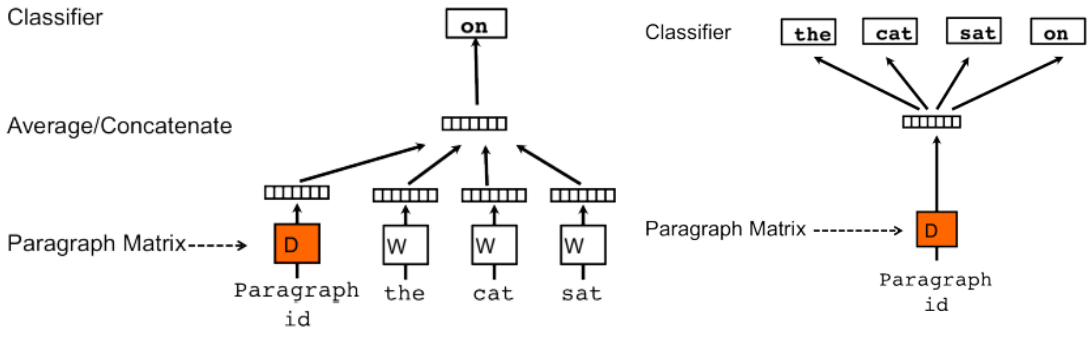
similarity. Multiple degrees of similarity are achieved by doing your comparison on a subspace of the overall vector space, for example the features you are interested in comparing. Lastly, because they are vectors in $\mathbb{R}^n$ we have a predefined addition and subtraction, the trick is then to get the desired result out of the operation by creating good vectors with the appropriate features.

These vectors are created using a combination of a **Continuous Bag-of-Words Model** and a **Continuous Skip-gram Model**. The Continuous Bag-of-Words Model uses a Feedforward Neural Net Language Model which consists of input, project, hidden, and output layers. Its job is to produce a feature vector based on the context that the word appears in. The Continuous Skip-gram Model is a Recurrent Neural Net Language Model which has the opposite job: to predict the context or words around based on the current word [11]. Word2Vec's Continuous Skip-gram Model also aggressively subsamples frequent words such as "the" and "a" during training in order to create better more accurate vectors for rare words [12].

Doc2Vec

Getting a semantic representation of words using Word2Vec is incredibly interesting, but generally in Information Extraction, we are not concerned with extracting individual words, but rather relations and facts. To achieve this, we need a semantic representation of the relation or fact in much the same way we did with words in Word2Vec. Building on the idea of vector representations of words, we can also represent a piece of text anywhere from a phrase to a full document as a vector using a training process similar to that of Word2Vec which can be seen in Figure 2.4a [8].

Formally called the Distributed Memory Model of Paragraph Vectors (PV-DM), the technique strives to create a feature vector which represents the context or topic of the paragraph itself in order to aid in the prediction step when attempting to predict words in a document as is being illustrated in Figure 2.4b [8]. Doc2Vec also allows you to add "tags" to your documents which do not have to be unique like your Paragraph Vector IDs, which enables you to tag certain documents and cluster them based on their feature vectors. Using this, we can then classify unlabeled documents by determining which of the tags its feature vector is closest to.



(a) Training in Doc2Vec

(b) Predicting in Doc2Vec

Figure 2.4: Doc2Vec Algorithms [8]

2.2 Computational Fact-Checking

The ubiquity of the Internet has led to a proliferation of information, but not all of it good. Due to a variety of factors, including the uncontrolled nature of the Web, the low barrier to entry for social media platforms, and the ability of a single individual to hold multiple accounts, it is easier than ever to disperse unreliable information [19]. Unfortunately, we are currently in the Information Age and just as information has revolutionized the world and been responsible for billions in profits, we are now seeing that misinformation can have an effect of the similar magnitude. In recent years, we have seen individuals and organizations spread misinformation to make profits, alter public opinion for financial or political gain, exert influence for a cause, and defame public figures [19]. As an example, during the 2016 United States Presidential Election, researchers were able to identify a network of agents on Twitter spreading misinformation which is visualized in Figure 2.5 [17]. Journalists and fact-checking websites have been working to check the factual validity of statements for years, but as the volume of information and the number of agents which we consider "sources" of information both rise, there is no way their methods can keep up.

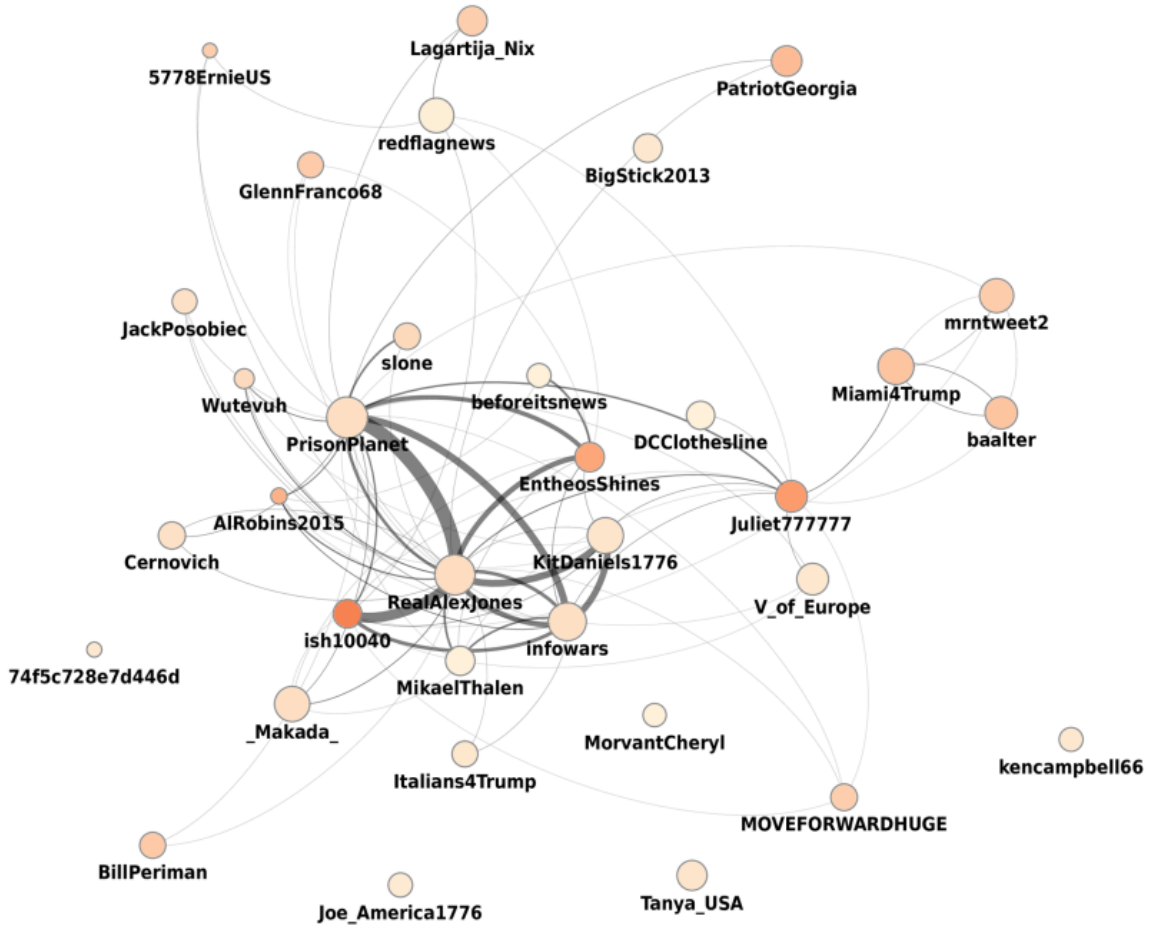


Figure 2.5: Retweet network of the core of spreaders of articles from low-credibility sources. Node size represents out-degree (number of retweeters) and node color represents in-degree. [17]

Determining the degree of truth of a claim is an extremely complex task. Even if this can be accomplished, one needs to consider the context of the claim to decide if despite being factually accurate, the claim misrepresents the bigger picture. This is a common

practice in finance and politics, where statistics or numbers are *cherry-picked* in order to show the company or politician in the best possible light, even if the best possible light is the only bright spot. Fact-checking then is the process of putting a claim into context, gathering relevant information, conducting thorough analysis, and reporting a conclusion with explanations and evidence. The amount of time and work it takes to accurately and thoroughly fact-check a claim leads to misinformation having anywhere from minutes to days to spread before any corrective action can be taken. In order to increase the efficiency of fact-checking the field of computational journalism has sprung up, hoping to tackle problems in journalism with Computer Science tools such as natural language processing, information extraction, data integration, and information visualization [19].

2.2.1 Knowledge Graphs

Creating a system for automated fact-checking represents a huge computational challenge, but recent advances in question-answering systems such as IBM Watson and PowerAqua not only give us hope, but also an avenue for exploration [19]. Those systems are able to tackle question-answering because they use a data structure known as a *knowledge graphs*. A knowledge graph G is an ordered pair $\mathbf{G} = (\mathbf{E}, \mathbf{R})$ where $\mathbf{E}$ is a set of entity or concept nodes and $\mathbf{R}$ is a set of relation or predicate edges. Within these systems, a *statement of fact* is represented as a subject-predicate-object triple where the subject and object are concepts and the predicate describes the relationship between the two concepts [2]. When a set of these facts are combined, we get a knowledge graph or semantic network.

Recent interest in curating knowledge graphs sprung out of the concept of a "semantic web" [15]. The World Wide Web Consortium (W3C) envisions an Internet which is a machine-readable "web of data," requiring linked data in a standardized format [15]. Parts of their vision has started to come together with the W3C Resource Description Framework (RDFs) gaining traction for publishing interlinked data on the web and although they have not tamed the entire Internet yet, they have provided useful standards and concepts for a resurgence in interest in knowledge graphs [19]. One such innovation is the introduction of Uniform Resource Identifiers (URIs) in their RDFs which allows a user to specify a relationship between entities from heterogeneous sources and without the fear of namespace conflicts [19].

The information that ultimately becomes the statements of fact in a knowledge graph can be gathered in a variety of ways, but the accuracy of the information greatly affects the usefulness of the final product. There are generally four approaches which all have a trade-off of efficiency versus accuracy [15]:

1. Curated approaches which feature experts manually constructing triples
2. Collaborative approaches which feature a group of volunteers manually constructing triples
3. Automated semistructured approaches which automatically extract information from semistructured sources using hand-crafted rules
4. Automated unstructured approaches which use Natural Language Processing and Machine Learning techniques

As the technique is less supervised, you have a higher probability of incorrect facts, whereas experts in a field carefully curating subject-predicate-object triples are much less likely to get it wrong than an algorithm using a pre-constructed set of rules and Named Entity

Recognition. Once we have the information in the form of triples in a well-defined ontology, there are a variety of tools and techniques from Graph Theory, Network Science, and Relational Machine Learning to manipulate and interpret it.

Properties of Knowledge Graphs

Meta information about the hierarchy of the expected data and type relations, called an *ontology* or *schema*, can also be captured in a knowledge graph using RDF Schema (RDFS) which is a language for expressing ontological facts [19]. A knowledge graph with instance data and ontological knowledge can be seen in Figure 2.6. This can be extremely important for contextualizing and reasoning about your data. For example, suppose you have the fact ("Barack Obama", "was", "President of the United States") in your knowledge graph. It would be extremely useful to know that all Presidents of the U.S. are politicians, and all politicians are humans, etc. This kind of type-based information is extremely helpful when you try to use your knowledge graph for automated reasoning, such as in the case of computational fact-checking. For example, given the statement of fact ("Banana", "is", "President of the United States"), an algorithm can easily determine this is not likely to be true as all other entities with the relation "is"/"was" to "President of the United States" were politicians and humans.

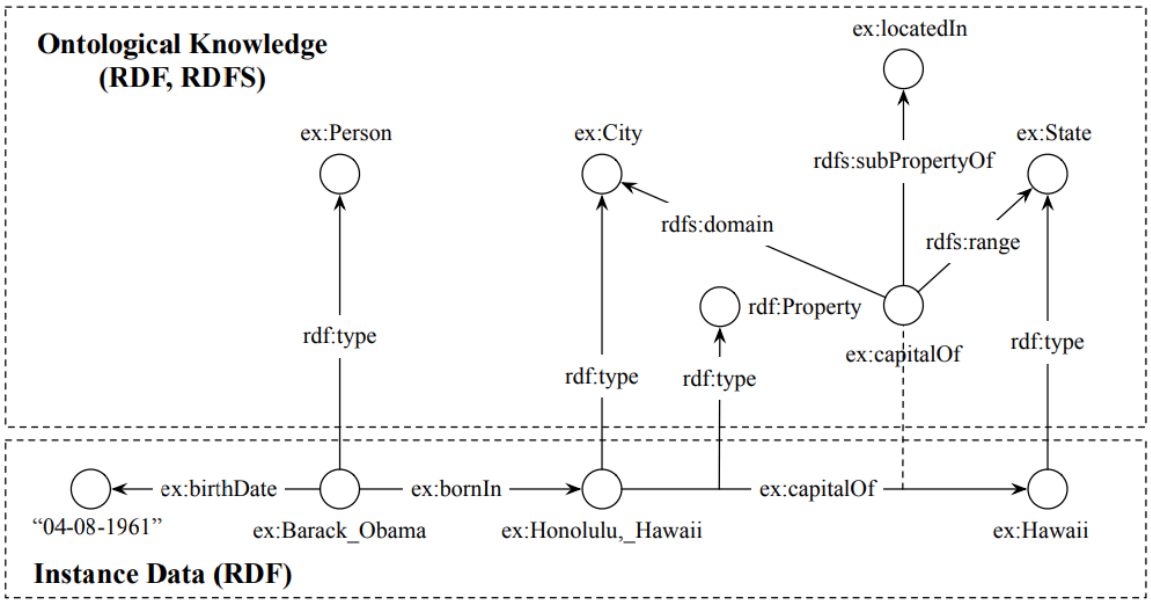


Figure 2.6: A knowledge graph with ontological information [19]

The graph structure of the data amplifies the benefits of the hierarchical structure of the ontology through transitivity [15]. We can infer new facts about the entities in our graph using the relations between the entities and the type constraints. As an example, if we have know ("Mr. X", "was born in", "Boston") and ("Boston", "city in", "United States"), we can infer that ("Mr. X", "was born in", "United States"). Besides concrete mathematical theorems, there are a variety of statistical patterns in network data that we are able to exploit as well.

Homophily is the tendency of individuals to associate and bond with others who are similar and is one of the biggest contributors misinformation spreading across the Internet. In social networks, it leads to groups of users called echo chambers which do not appreciate

contrary views or a diversity of opinion [19]. However, it can also be used to our advantage when performing relational learning. This is helpful for us because we can cluster entities based on links and from there infer links based on the what is known about similar entities. Social media does this when they realize that you and 30 other people all like the same 10 bands, so the social media platform recommends a band to you that the other 30 people also all like.

Network structures also tend to be able to be divided up into "blocks" which are sets of entities such that all of the members of the set have similar relationships to the members of another set [15]. This is analogous to, a bipartite matching. An example of two blocks would be "Science Fiction writers" and "Science Fiction franchises". All entities in the "Science Fiction writers" block would have the same relation ("authored") to an entity in the "Science Fiction franchises" block.

Relational Machine Learning on Knowledge Graphs

Graphs can usually be represented as matrices, called adjacency matrices, which hold information about whether or not two nodes or vertices in a graph are adjacent, meaning they have an edge between them. The knowledge graph analog, because of the addition of specific relations on the edges is an adjacency tensor [15]. Let $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ be the set of all concept nodes in $\mathbf{G}$ and $\mathcal{R} = \{r_1, r_2, ..., e_{|\mathcal{R}|}\}$ be the set of all relation tyes in $\mathbf{G}$. Each possible statement of fact, (i, j, k) where i, $k \in \mathcal{E}$ and $j \in \mathcal{R}$ can be grouped together to form a $\mathcal{E} \times \mathcal{R} \times \mathcal{E}$ third-order tensor $\underline{\mathbf{Y}}$ where $y_{ijk} = 1$ if the triple (i, j, k) exists and 0 otherwise [15]. We can also model each unknown triple x_{ijk} as a binary random variable with each possible realization of $\underline{\mathbf{Y}}$ representing a possible world [15].

The task of learning new relations between the entities in your graph then becomes the task of estimating a probability distribution $\mathrm{P}(\underline{\mathbf{Y}})$ from a subset of the relations $\mathcal{D} \subset \mathcal{E} \times \mathcal{R} \times \mathcal{E} \times \{0, 1\}$ of triples already present [15]. It is worth nothing though that the adjency tensor scales extremely quickly as the number of possible triples is $|\mathcal{E} \times \mathcal{R} \times \mathcal{E}|$. Type constraints allow us to reduce the number of possible triples considerable to the set of syntactically valid triples, but even from there only a small fraction is likely to be true [15]. The sparsity of the relations and the $\mathcal{O}(\mathcal{R} \times \mathcal{E}^2)$ size of the adjacency tensor mean that methods for relational learning on large knowledge graphs have to be done very efficiently to have any real impact.

One way to systemically target relations is to use the correlations between the relational data. The absence or presence of data in a knowledge graph is correlated and therefore predictive of the presence or absence of other triples due to their relational nature. Mathematically, this means that the random variables $y_{ijk} \in$ 0,1 (the possible triples) are correlated with each other. There are three main models for correlations between subject-predicate-object triples: Latent Feature Models, Graph Feature Models, and Markov Random Fields [15].

The Latent Feature Model assumes that all y_{ijk} are conditionally independent given latent features associated with subject, object and relation type and additional parameters. A latent feature is something not directly observable in the data, but that can be inferred. These models use a latent feature representation of entities, so for any entity $e_i \in \mathbf{E}$ there would be a feature vector representation $\overrightarrow{e_i} \in \mathbb{R}^n$ where n denotes the number of latent features in the model. In this model, the relationships between entities are assumed to be derived from the interactions of their latent features [15].

Graph Feature Models assume all y_{ijk} are conditionally independent given observed graph features and additional parameters. Rather than depending on features of the data,

this approach looks directly at the structure of the graph, hoping to infer information by extracting features from the observed edges in the graph. As an example, it is often the case that parents of a person are married, so a graph feature model may predict the triple ("John", "married to", "Mary") based on the existence of a common child in the graph [15].

Latent Feature Models and Graph Feature Models predict the existence of a triple x_{ijk} using a score function $f(x_{ijk}; \Theta)$ where Θ is the parameters and f gives the model's confidence in the existence of the triple x_{ijk}. Due to the conditional independence assumptions, the probability model for both of these models is

$$P(\underline{\mathbf{Y}} \mid \mathcal{D}, \Theta) = \prod_{i=1}^{|\mathcal{E}|} \prod_{j=1}^{|\mathcal{E}|} \prod_{k=1}^{|\mathcal{R}|} \mathbf{Ber}(y_{ijk} | \sigma(f(x_{ijk}; \Theta))) \tag{2.2}$$

where σ is the is the sigmoid or logistic function ($\sigma(u) = 1/1+e^{-u}$) and **Ber** is the Bernoulli distribution ($\mathbf{Ber}(y|p) = p$ if y = 1, 1-p if y = 0) [15].

Markov Random Fields assume all y_{ijk} have local interactions, so it drops the assumption that the random variables y_{ijk} are conditionally independent and claims that each y_{ijk} can depend on any of the other $|\mathcal{E} \times \mathcal{R} \times \mathcal{E}| - 1$ random variables in $\underline{\mathbf{Y}}$ [15]. The Markov Random Field method uses a dependency graph with random variables as the nodes and statistical dependencies as the endges to encode dependencies between random variables, and using these dependency graphs, we can say that

$$P(\underline{\mathbf{Y}} \mid \theta) = \frac{1}{Z} \prod_{c} \psi(y_c | \theta) \tag{2.3}$$

where $\psi(y_c|\theta) \geq 0$ is a potential function on the cth subset of variables (the cth clique in the dependency graph) and $\mathrm{Z} = \sum_y \prod_c \ \psi(y_c|\theta)$ is the partition function that makes the distribution sum to one [15].

2.2.2 Computational Fact-Checking from Knowledge Graphs

> *If a Lie be believd only for an Hour, it has done its Work, and there is no farther occasion for it. Falsehood flies, and the Truth comes limping after it.*
>
> -Jonathan Swift [18]

As we have seen, the problem of filtering misinformation out is both a crucial and challenging one, but how can it be accomplished? This is a burgeoning area of research given recent events. Promising and innovative techniques such as those proposed by Wu et al. check the sensitivity of a claim by examining how the strength of the claim changes as the parameters of the claim vary slightly [23]. The vast amounts of relational information available and the relational tools available for manipulating them has lead many to search for answers through knowledge graphs. The problem then becomes: given a knowledge graph populated with true facts, how can one discern if a new fact is true or false?

The transitivity of relations between entities in a knowledge graph is a powerful tool, but it has its limitations. There must be a law of diminishing returns when we say that an entity x is related to another entity z because both x and z are related to y. If there is not, we quickly get complete connected components, although we know that the true number of correct relations is sparse. Thus, when considering relating entities through transitivity, we take into consideration the path length.

Knowledge Linker and Relational Knowledge Linker

In a knowledge graph, there may be many paths from a subject to an object even of the same path length, but each of them provides different factual support for their relation by the chain of entities and relations which bring them together. As an example, paths containing generic concepts such as "Male" are less likely to provide specific support for our target subject and object being related than paths containing only specific concepts. To formalize this intuition, this is called the **semantic proximity** and is defined as

$$\mathcal{W}(\mathbf{P}_{s,o}) = \mathcal{W}(v_1, ..., v_n) = \left[1 + \sum_{i=2}^{n-1} log\ k(v_i)\right]^{-1} \tag{2.4}$$

where $k(v)$ is the degree of entity v. Defining the truth value of a relation $\mathcal{T}(s, p, o) \in$ [0,1] to be max $\mathcal{W}(\mathbf{P}_{s,o})$ gives a truth values which maximizes the semantic proximity, is equivalent to finding the shortest path, and provides the maximum information content. This approach to Computational Fact-Checking is called Knowledge Linker and its process is shown fact-checking the statement "*Barack Obama is Muslim*" on Wikipedia data in Figure 2.7 [2].

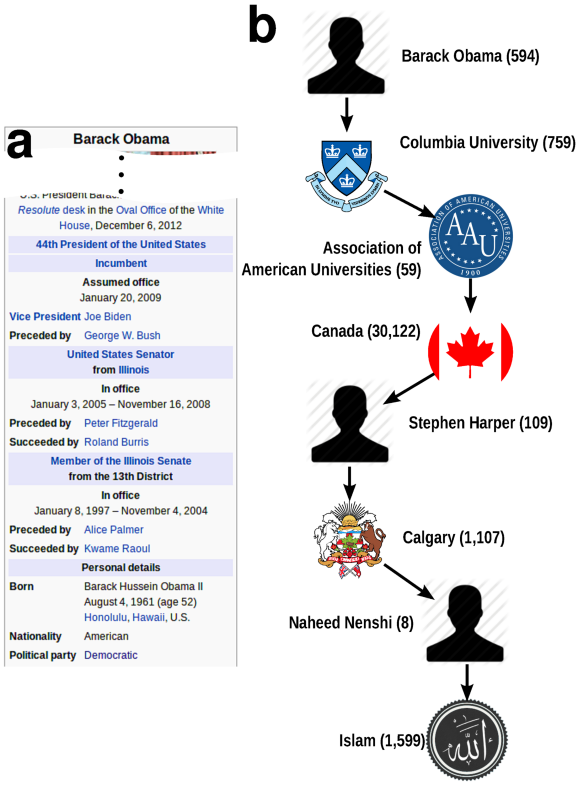


Figure 2.7: A path from Barack Obama to Islam containing the very non-specific entity, Canada. The numbers in parentheses denote the degree of the node. [2]

This approach was extended and improved on by capturing the semantic similarity of the relations along the path, an approach known as Relational Knowledge Linker. This is achieved using the line graph of the knowledge graph. Letting $\mathcal{R}$ be the set of the types of relations in the graph, the adjacency matrix of the line graph of a knowledge graph, $\mathbf{C}$, is a $|\mathcal{R}| \times |\mathcal{R}|$ matrix representing the co-occurrences of $\mathcal{R}$. In order to not have the

matrix dominated by the most common relationships, Term Frequency-Inverse Document Frequency is then applied to C to produce C' using the following equations: [20]

$$\begin{aligned} TF(r_i, rj) &= log(1 + C_{ij}) \\ IDF(r_j, \mathcal{R}) &= log\Big(\frac{|\mathcal{R}|}{|\{r_i | C_{ij} > 0\}|}\Big) \\ C'(r_i, r_j, \mathcal{R}) &= TF(r_i, r_j) \cdot IDF(r_j, \mathcal{R}) \end{aligned} \tag{2.5}$$

The rows of C' are then treated as feature vectors of the relations they represent. The relational similarity, u, between relations r_i and r_j, is defined as $u(r_i, r_j)$ equals the cosine similarity of the i-th and j-th rows of C'. With this definition of relational similarity, we can replace the notion of path specificity laid out in Equation (2.4) with:

$$S'(P_{s,p,o}) = \left[\sum_{i=2}^{n-1} \frac{log\ k(v_i)}{u(r_{i-1}, p)} + \frac{1}{u(r_{n-1}, p)} \right]^{-1} \tag{2.6}$$

where again k is the degree of the node [20].

Knowledge Stream

A single path may not capture the true semantic similarity between the two target entities however, and as previously stated a large part of fact-checking is putting the fact into its broader context. The former approach also ignores the relations along the path and the type of entities, only taking into consideration the degree of the node. Another approach to computational fact-checking on knowledge graphs considers a larger subset of the graph as its context and uses the similarities between the ontological relations to help determine which paths are more semantically meaningful for the target relation.

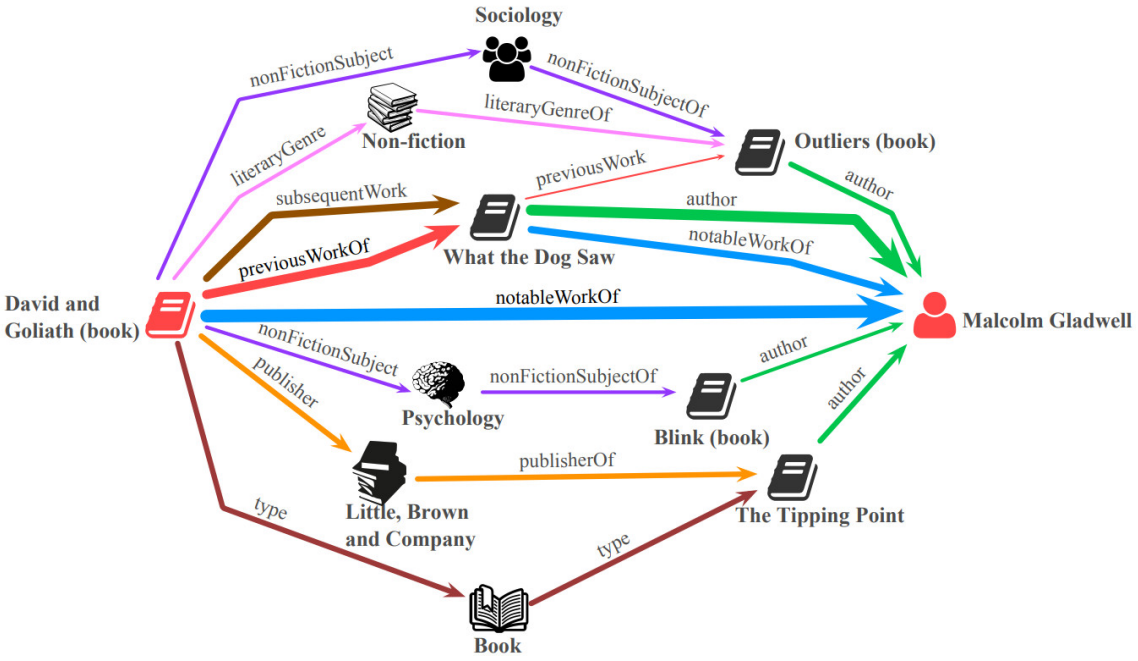


Figure 2.8: Finding Streams in Knowledge Graphs [20]

The approach is called Knowledge Stream because it treats computational fact-checking as a network flow problem, determining how much information it can push from subject to object [20]. It labels the relational edges along the way with capacities as a function of how semantically similar they are to the target relation and costs to traverse nodes which are a function of the degree of the node [20]. A visualization of Knowledge Stream or *KSTREAM* can be seen in Figure 2.8

More specifically, the lower bound of each edge (the required flow) is zero and the upper bound (capacity) of each edge $e = (v_i, v_j)$ for subject-predicate-object triple (s, p, o) is

$$\mathcal{U}_{s,p,o}(e) = \frac{u(g(e), p)}{1 + log\ (k(v_j))} \tag{2.7}$$

where u is the relational similarity between the edge label $g(e)$ and the target predicate p and k is the degree of the node [20]. The cost of each edge $e = (v_i, v_j)$ is the generality of the object node $log(k(v_j))$ [20]. With these constraints, the problem is then turned into a minimum cost maximum flow problem, which is a commonly studied problem in computer science and the solution is optimized using Successive Shortest Path on the residual network [20].

PredPath

Our last method, PredPath, attempts to learn what examples of the target relation looks like from the graph itself and then determines if the target relation fits in with what it has learned. It abstracts the subject and object and tries to understand what it means for the more generalized subject and object to be related by the target relation, much like a human fact-checker would. Given the statement "*Chicago is the capital of Illinois*," the model abstracts *Chicago* to *U.S. City* and *Illinois* to *U.S. State* then tries to understand what it means for a *U.S. City* to be *capitalOf* a *U.S. State* [18].

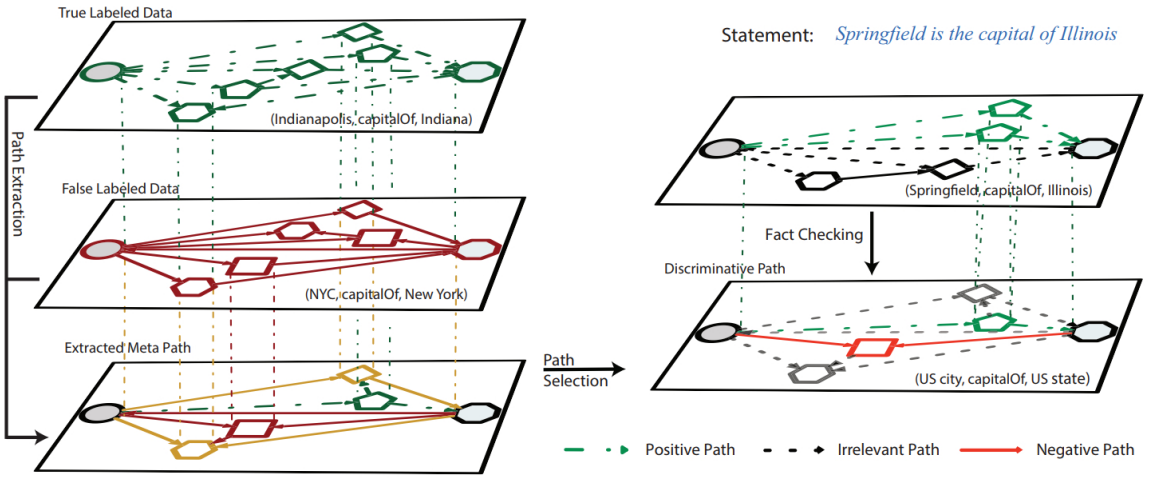


Figure 2.9: An overview of how PredPath works [18]

The method begins by specializing the concept of a path to knowledge graphs. A **Meta Path** on a knowledge graph $\mathbf{G}$ is a directed, typed sequence of vertices and edges $P^k = o_1 \xrightarrow{p_1} o_2 \xrightarrow{p_2} ... \xrightarrow{p_{k-1}} o_k$ in $\mathbf{G}$ where o_i denotes nodes in $\mathbf{G}$ and $\xrightarrow{p_i}$ denotes an edge in $\mathbf{G}$. An example of a meta path could be {U.S. City}$\xrightarrow{headquarter}^{-1}${State Agency}$\xrightarrow{jurisdiction}${U.S. State} [18].

Define an **Anchored Predicate Path** of length k to be a directed, typed sequence of edges with typed-endpoints $P^k = o_1 \xrightarrow{p_1} \xrightarrow{p_2} ... \xrightarrow{p_{k-1}} o_k$ [18]. Essentially this is a meta path, without the intermediate nodes. Let a set of **Discriminative Paths** of length less than or equal to k be defined as a set $\mathbf{D}^k_{(o_u,o_v)}$ of anchored predicate paths that alternatively describe the give statement of fact $o_u \xrightarrow{p} o_v$ with maximum path length k [18].

PredPath views fact-checking as a supervised link prediction task; that is, given a subject-predicate-object triple (s, p, o), the algorithm discovers if the fact is implied by the information already in the graph. This is done by generating a set of positive training examples $\mathbf{T}^+ = \{(u, v)|u \xrightarrow{p} v \in \mathbf{G}\}$ and negative training examples $\mathbf{T}^+ = \{(u, v)|u \xrightarrow{p} v \notin \mathbf{G}\}$ where u and v have the same type, connectivity, etc. as s and t respectively [18]. Once the training is complete, the algorithm gives a prediction score for the given statement. PredPath's process is for "*Springfield is the capital of Illinois*" can be seen in Figure 2.9.

Chapter 3

Proposed System Architecture & Research

> Processed data is information.
> Processed information is knowledge.
> Processed knowledge is Wisdom.
>
> Ankala V Subbarao

Motivated by the problem of unstructured data abundancy on the Internet, and manual information access by the analysts at Praedicat Inc., we started working towards the problem of automating data extraction and performing information aggregation on unstructured data for business company profiling. To achieve this task, the research problem was divided into three components:

1. **Web Crawling Framework** (*From data to information*): The modules contained in this component regulate the extraction of data from various sources on the Internet, and parse them into text formatted information.

2. **Information Classification & Aggregation** (*From information to knowledge*): The textual information extracted using the Web Crawling Framework is pipelined into the Classification and Aggregation modules, which filter and prioritize the information into a master document for the analysts.

3. **Computational Fact-Checking using Knowledge Graphs** (*From knowledge to Wisdom*): Even though, we were able to accumulate usable knowledge, the system was not yet intelligent to act upon that knowledge. To tackle this, we utilized computational fact-checking algorithms, which would allow the analysts to query a knowledge graph about the veracity of triples-based statements.

Further, Fig. 3.1 illustrates these components and submodules. The proposed system architecture takes in a company name or a unique identifier for a company and formulates a query on the input. This query is transformed into various formats and passed to different web-crawling modules which return relevant web-links. These web-links are parsed by general and specialized parsers to return raw textual information. The raw text is passed onto a classifier which filters out spurious information. The information is prioritized and maintained in a master document for each query by the profile manager. The raw textual information would then be transformed into RDF triples [Section 2.2.1] and merged into

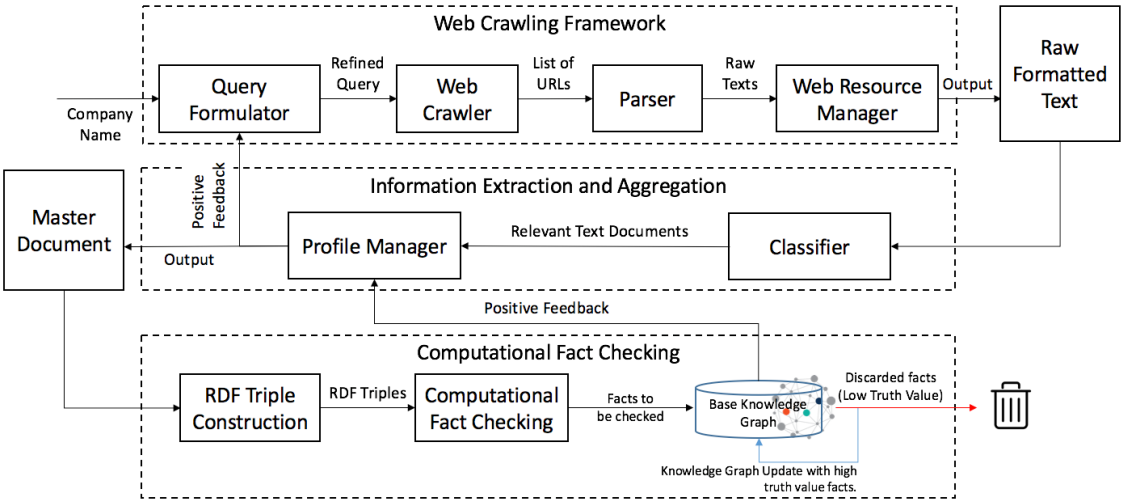


Figure 3.1: Overview of System Architecture

the knowledge graph. The knowledge graph is then used to computationally fact-check truthfulness of new triples and recursively add them to the graph itself. Once, the knowledge graph is updated, it can be used to fact-check new tuples creating a positive feed-back loop. In the following sections, we discuss each of these procedures in detail.

3.1 Web Crawling Framework

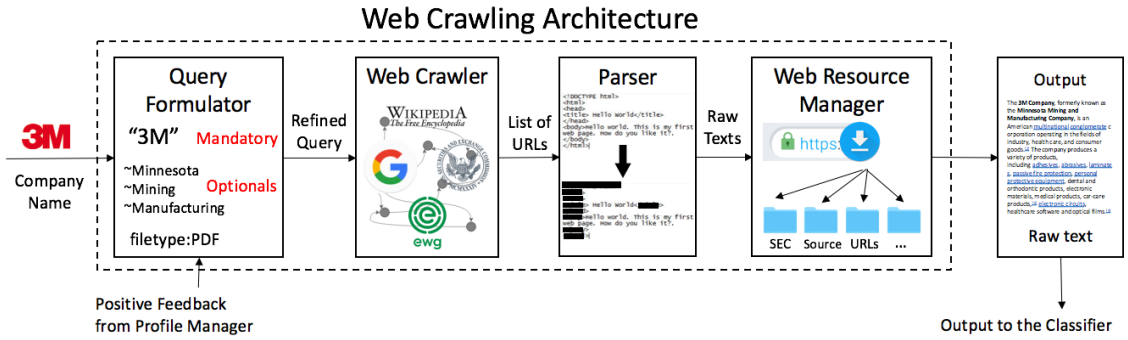


Figure 3.2: Detailed architecture for the Web-Crawling Framework

The web-crawling framework (Fig. 3.2) contains the *Query Formulator* (Sec. 3.1.1), the *Web Crawler* (Sec. 3.1.2), the *Parser* (Sec. 3.1.3) and the *Web Resource Manager* (Sec. 3.1.4). The first module i.e. the *Query Formulator* takes in a query and the final module i.e. the *Web Resource Manager* returns a set of raw text data which is passed onto the classifier (Sec. 3.2.1).

3.1.1 Query Formulator

The web-crawling framework is initialized with a company name or an ID (Central Index Key (CIK) code) provided by an analyst. This input is taken by the *Query Formulator*. The

purpose of the *Query Formulator* is to refine the initializing query to make them pertinent to the information we search for. A simple example of an unrefined query could be *"Apple"*, when the purpose is to search information relevant to *"Apple Inc."*. The query *"Apple"* will return search results which might be more closely related to the apple fruit rather than the company. This problem of ambiguity needs to be tackled before we can land on relevant search results. The *Query Formulator* achieves that by making specific keywords as mandatory and aliases as optionals. Further, we specify the *filetype* to retrieve only specific kind of document files. For Praedicat Inc., the *Query Formulator* aims to form an effective query that links a company to its potential business activities.

3.1.2 Web Crawler

An efficient query lands us onto relevant URLs. The *Web Crawler* automates the process of visiting, indexing and storing these URLs. However, we cannot aimlessly hover around the Internet and hope to find the right information. An intelligent web crawler should know where to crawl data from and how deep to go into each data source. In other words, it needs to know which links to traverse and how much to traverse any particular web-link. This is done by setting an upper bound on the maximum depth of traversal and performing a first-level elimination of spurious links. Due to the high variability of website directory structures, we tailored website-dedicated web crawlers to get relevant links. We were able to create high credibility databases for Praedicat Inc.. We designed several web crawlers tailored to specific sites to get relevant links, some of which are listed below:

1. A web crawler for the **U.S. Securities And Exchange Commission**[1] that can get links to Form 10-K, the annual report of the company's business and financial condition that includes audited financial statements, Form 8-K, the quarterly report, and Form EX-21, the overview of subsidiaries of a company.

2. A web crawler for **Wikipedia**[2] that can get the links to companies' Wikipedia page,

3. A web crawler for **Google**[3] that can get links to search results and links to a company's subsidiary information,

4. A web crawler for the **Toxics Release Inventory(TRI) Program**[4] that can get links to a company's facility report and chemical usage,

5. A web crawler for **EWG Skin Deep Database**[5] that can get links to all cosmetic products manufactured by a company and all ingredients used in a product,

6. A web crawler for **National Pesticide Information Retrival System (NPIRS)**[6] that can get links to pesticide manufacturers that use a certain chemical.

3.1.3 Parser

Once a list of relevant URLs is generated by the *Web Crawler*, it is pipelined to the *Parser*. The *Parser* transforms the contents of the webpages to text documents that are human-

[1] `https://www.sec.gov/`
[2] `https://www.wikipedia.org/`
[3] `https://www.google.com/`
[4] `https://www.epa.gov/toxics-release-inventory-tri-program`
[5] `https://www.ewg.org/skindeep/`
[6] `http://npirspublic.ceris.purdue.edu/ppis/`

readable and analyzable via natural language processing techniques. Since, (1) the text is formatted in various ways and (2) not all of the text information on a webpage is desirable (e.g. advertisements, headers and footers), a parser should be generalized, such that it can be used to parse highly variable data crawled by the web-crawler. We designed our parser to retrieve visible text on the web pages. The parser also contained filters to scrap out irrelevant social media links and advertisements. Further, we designed specific parsers for high credible information extracted via the web-crawlers to prevent information loss. The various text parsers designed are enumerated below. Further, the raw text information produced by these parsers and various challenges encountered are described in the Results (Chapter 4).

1. A parser for **companies' Wikipedia pages** that can parse information in a Wikipedia infobox into a Python Dictionary and the article text as a string.

2. A parser for **all-level subsidiaries returned by Google** that can parse subsidiary names on a search result page into a Python Dictionary.

3. A parser for **TRI facility reports** that can parse a facility information table into a Python Dictionary and a chemical usage report into a comma-separated values (CSV) file.

4. A parser for **EWG search results** that can parse name of products by a company and ingredients in a product into a Python Dictionary.

5. A parser for **NPIRS search results** that can parse names of manufacturers that use a certain hazard into a Python Dictionary.

6. A general parser that is able to get visible texts on a web page that filters out irrelevant information such as social media links and advertisements.

The motivation to design specific parsers originates from the idea of designing the system architecture as a Positive Feedback Loop (Sec. 3.2.2). A credible information feedback from the *Profile Manager* back to the *Query Formulator* which can use this information to build better queries, positively reinforce the subsequent modules, generate more relevant results and aid the architecture to move from structured to unstructured information.

3.1.4 Web Resource Manager

The *Web Crawling Framework* is capable of producing a large volume of data, but it is important that the information is managed in a way that it is useful. A common theme for companies with a lot of data is that they cannot make use of it. Even though our data is represented many ways (HTML, text, lists of words, vectors, etc.), it is important that it is always traceable back to the source as well as accessible and thus queryable. It needs to be traceable back to the source so we can justify the conclusions we make with tangible evidence rather than simply pointing to the output of a system. Aggregating this information in a queryable format makes it usable for query formulation and building company profiles. The complexity of the architecture and variety of data types (chiefly HTML, PDF, text, URLs) render most traditional data storage solutions incapable of helping us. The variety of our data, integration needs, and the desire to have random access to source documents mean that we had to make our own data storage solution, but our solution still needs to fulfill the

traditional role of a data management solution: a uniform as possible entries, queryability, and efficiency.

Web Resource Manager was designed to meet these needs for a database of web resources. Each data entry is given a UUID (Universally Unique Identifier) and the instance saves a mapping from URL to UUID. Both the text and source for each web resource is saved using this UUID. The source document (HTML or PDF) is saved in "source/⟨resource_uuid⟩.(pdf/html)" while the text information, UUID, generating query, and URL is saved in a JSON file [7] at "docs/⟨resource_uuid⟩.json." Using this uniform data storage system and a simple API. [8] *Web Resource Manager* makes storing and querying the contents and source files of web resources trivial.

3.2 Information Classification & Aggregation

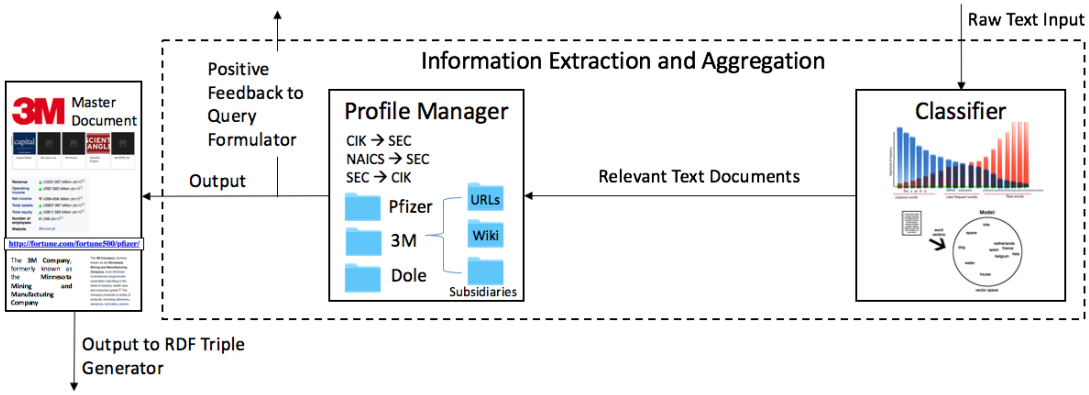


Figure 3.3: Detailed Architecture for Information Classification and Aggregation

As one would imagine, the Internet contains a lot of information that our *Web Crawler* and *Parser* is able to retrieve, but the grand majority of it is not useful information. The *Query Formulator* is able to help target our search in some regards, but among the web pages that contain valuable information, this information will usually be sparse compared to the amount of "filler." This paragraph itself is a perfect example: it creates context, motivates the section, and helps with flow so we don't just throw you into technical details, but it doesn't contain contain any new facts that an Information Extraction system would be looking for. Lastly, when we have the set of "relevant" information we need to aggregate the information in a useful format.

3.2.1 Classifier

Our goal with the *Classifier* is to create a classifier for taking the visible text on a website and deciding which information is irrelevant meaning it can be discarded or relevant meaning

[7] In computing, JavaScript Object Notation or JSON is an open-standard file format that uses human-readable text to transmit data objects which are serializable.

[8] In computer programming, an application programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication between various components.

it can be passed along to be aggregated for end-users and fed into the rest of the system. There are a variety of challenges associated with this. One needs to identify when the text is truly the text from the website that the *Query Formulator* identified as a target or text notifying you that the site is down, that you've been flagged for violating terms of usage, or a redirect notice. Even if you get the correct text, there will be "junk" text such as the navigation bar, ads, and the footer. Additionally, the domain of the request of the target text can matter a lot as a general-purpose classifier might target "Javascript" as an Internet keyword associated with "junk" text i.e. "*Ensure that Javascript is enable to view this page*," but an Information Extraction system looking for facts on programming languages would be led astray by this rule.

Our *Classifier* went through three iterations each with a different algorithm for classification. The tools and techniques we used for the *Classifier* will be discussed here, with a focus on the strengths and weaknesses of each model with more in-depth analysis for the model we settle on in Chapter 4 which will cover our Results. As a general rule for all of the models, we used the sentence as the atomic unit. This is because facts and relations cannot be captured at the word level, but classifying entire documents seemed like a poor choice as documents can contain good information surrounded by "junk."

Classification Algorithm 1: Keyword-Based Classification

Our first attempt at a classification algorithm was rather simple, using a set of words and phrases that we identified as being generally associated with "junk" text and removing all of it. An example of one such phrase would be "*Terms and Conditions.*" We believe that sentences from web sources containing the phrase "*Terms and Conditions*" are much less likely to contain meaningful information and much more likely to contain "junk" text such as disclaimers. "*Contact Us*" is another such phrase which is more likely to be associated with footer, header, or navigation bar text rather than meaningful information.

The problem with this approach is that it is not very robust. We did not have a data set with labeled with examples of good or bad text which meant that we had to make the list of keywords using our observations and intuition rather than a statistical approach. Further, the approach doesn't easily support scalability or improvement. Lastly, the approach's binary decision of "bad" is narrow because the existence of a "bad" keyword in the sentence only takes one feature of the sentence into consideration.

Classification Algorithm 2: Term Frequency-Inverse Document Frequency

Term Frequency-Inverse Document Frequency (TF-IDF) increases proportionally to the number of times a word appears in the document and is offset by the frequency of the word in the overall set of documents [9]. The term frequency (tf) part describes the number of times each word appears in a document divided by the number of times it appears in all of the documents where $tf_{i,j}$ is the number of occurrences of term i in document j [9].

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \tag{3.1}$$

The Inverse-Document Frequency part gives a number proportional to the inverse of the proportion of documents containing the term where df_i is the number of documents containing i and N is the total number of documents [9].

$$idf(w) = log(\frac{N}{df_t}) \tag{3.2}$$

We are looking for the sweet spot of words that are not extremely common in all documents, but are not rare either. TF-IDF scores are [9]:

1. highest when the term occurs many times within a small number of documents

2. lower when the term occurs fewer times in a document, or occurs in many documents

3. lowest when the term occurs in virtually all documents.

This makes TF-IDF somewhat problematic for our application. Suppose I start the architecture with the query "Apple." A good query formulator combined with Google's page rank algorithm will result in a set of documents which more than likely all contain the word "Apple." Additionally, because we are dealing with proper nouns, it is commonplace that the frequency of the terms is not high enough to offset this because authors will use a variety of pronouns and aliases to avoid repetition. As an example, an article about *Apple Inc.* will generally refer to it as "the company" or "they." Words that did contribute to a high TF-IDF, ones that occurred many times within a small number of documents, in our case were words like "Javascript" which appeared in notices to ensure that our Javascript was enable in our browser.

Classification Algorithm 3: Self-Supervised Learning with Doc2Vec

Our proposed solution is to use a more robust version of the Keyword-Based Classification model using self-supervised learning. The idea draws inspiration from Open Information Extraction's revolutionary TEXTRUNNER which automatically labels its own training data for a Naive Bayes classifier which is then used by an extractor [1].

Along those same lines, we are using a set of rules to have the *Classifier* automatically label training data as "bad", training a Doc2Vec model tagging "bad" examples with the "bad" tag, and then computing similarity scores to the "bad" vector. This approach of automatically labeling web-based data and using Doc2Vec to score documents based on relevance will be referred to as *Web Resource Manager Doc2Vec Classifier*. We also tested the classifier on financial statements, namely the 8-K and 10-K information we gathered from the SEC, this time also labeling "good" examples using a list of companies provided by Praedicat, Inc. and the Chemical Abstracts Service list of chemicals which we call *Profile Manager Doc2Vec Classifier*.

As previously mentioned, we view the sentence as the atomic unit of semantic meaning at the classification level, so the first step is to break up a document into sentences using *sentence_tokenize* from *nltk*. Next, we tag the sentences that fit the characteristics of "junk" text such as the existence of certain keywords, anomalous character lengths, and other syntactic heuristics. Passing these training examples to Doc2Vec allows us to compare the latent features using the Paragraph Vector rather than simply making decisions based on these heuristics. This approach is also much easier to scale because we can query the model about which words are most similar to the current "bad" keywords in order to get insights as to what other words are characteristic of "junk" text.

In order to compute the similarity scores, we used the default similarity function in Doc2Vec which is cosine similarity. It is the standard measure used in information retrieval and calculates the cosine of the angle between two Euclidean vectors meaning it is unaffected by scalar transformations [22]. The cosine similarity of two vectors $\overrightarrow{x}$ and $\overrightarrow{y}$ can be computed as:

$$\frac{\sum x_i y_i}{\sqrt{\sum x_i^2}\ \sqrt{\sum y_i^2}} \tag{3.3}$$

By computing the similarity score of each sentence in a document to the "bad" tag (or "good" and "bad" tags and taking their quotient), we are able to rank the relevance of the sentences.

3.2.2 Profile Manager

ProfileManager was designed for the aggregation of information related to corporate entities to support building business profiles. It uses the United States Securities and Exchange Commission (SEC) Central Index Key (CIK) to act as universally unique identifiers (UUIDs) and allows the user to compile a variety of information on corporate entities in an easy to use and query format because each profile is a dictionary. Assisting the accessibility of information, *Profile Manager* includes a series of mappings from CIK codes to names and back, names to aliases, and mappings from industry codes (namely The North American Industry Classification System (NAICS) and Standard Industrial Classification (SIC) codes) and descriptions of them. The goal is to provide for a flexible data solution that allows us to accumulate all of the relevant information about a corporate entity in one place.

The *Profile manager* also generates a positive reinforcement of aggregated data as an output back into the *Query Formulator*. This input into the *Query Formulator* helps it to generate better queries and retrieve more relevant documents. To ensure that we never negatively reinforce the cycle, the data aggregated in the first iteration of the Positive Feedback Loop should be highly credible. To ensure the credibility of this information, we gather it from highly credible structured data sources which prevents any losses during parsing and classification. Once, we have an initial high credible database, it can be successively built through each iteration of the loop, eventually moving from structured non-indicative information (e.g. Government Filings, Facility Reports) to unstructured indicative information (e.g. News Sources).

3.3 Computational Fact-Checking using Knowledge Graphs

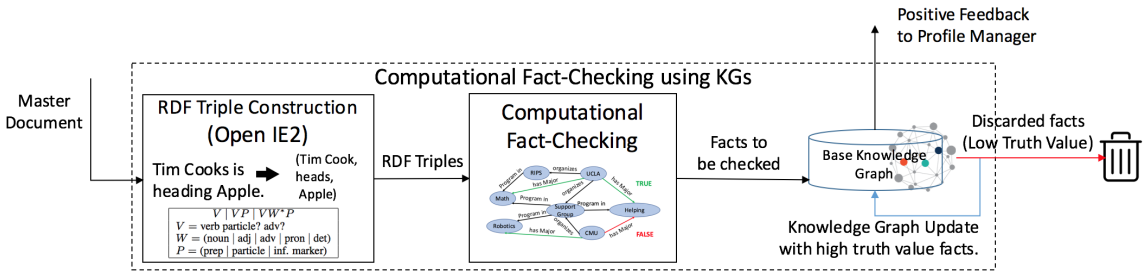


Figure 3.4: Computational Fact-Checking section of our Architecture

Even when information is found that seems relevant to our query, the information may not be factually accurate. Between fake news, astroturf, and opinions, the structure and

presentation of false information is diverse and dynamic, but also generally camouflaged so that even a human analyst is reliably able to distinguish between fantasy and reality. It isn't enough to ensure that we are finding *relevant* statements, we need to ensure that the models we are building are accumulating *accurate* statements to be useful.

To accomplish this, we need to look at more than the keywords, structure, or latent features of the information we are finding, we need to examine the logical structure of the statement and see if it contradicts our set of possible worlds. The logical structure can be extracted using Open Information Extraction as discussed in Section 2.1 and we can use the Computational Fact-Checking methods discussed in Section 2.2, but what do we check it against? There are some existing open-source Knowledge Graphs such as DBpedia[9] and YAGO[10], but these databases may not contain a lot of domain-specific information and may not be suitable for every application, which was the case for us meaning we first had to construct scrape structured and semistructured websites such as *SEC.gov*'s EDGAR database, *EPA.gov*'s TRI database, and a variety of others to construct a knowledge graph. These site-specific web scrapers and the information is discussed in more detail in Section 3.1.2.

Utilizing the approaches discussed in Section 2.2.2, we developed a novel algorithm *StreamMiner* which incorporates various components of PredPath [18], K-Linker [2] and Knowledge Stream [20]. The proposed algorithms is based on *PredPath's* ideology of construction of positive and negative examples of discriminative paths is the most intriguing because it is extremely intuitive, performing the task in the same way a human fact-checker would, and robust due its ability to "abstract" through the ontologies type hierarchy and "make analogies" through the discriminative paths. However, we believe that the approach is restricted because it uses an integer parameter to set the maximum length of the discriminative paths it will consider. Through thorough analysis of the use-cases we made modifications to the above algorithms was made.

1. Specificity over path length: Instead of using path length as a factor in determining the paths between object and subject, we use weight of total path specificity to mine path.

2. Fusing both node specificity and predicate similarity: We use the path specificity measure (Eq: 3.4) adopted from [20]. This metric would take into account the degree of the node and the relative similarity of each edge's predicate to the target predicate, giving preference to paths which have a closer semantic meaning to the target relation and paths which have nodes that are less general despite the absolute length of the path, which research has found to be a desirable property in Computational Fact-Checking [2, 20].

$$S'(P_{s,p,o}) = \left[\sum_{i=2}^{n-1} \frac{log\ k(v_i)}{u(r_{i-1}, p)} + \frac{1}{u(r_{n-1}, p)} \right]^{-1} \tag{3.4}$$

Extending our framework to include a Knowledge Graph and Computational Fact-Checking algorithms as seen in Figure 3.4 allows the system to perform state-of-the-art automated logical classification of information and helps in directing the search of the system. Relevant information can be converted to subject-predicate-object triples using Reverb[11] which then allows us to check the factual validity of each statement in the relevant information gathered.

[9] `https://wiki.dbpedia.org/`

[10] `https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`

[11] `http://reverb.cs.washington.edu/`

The knowledge graph can then be updated with high-truth value triples and low-truth value triples can be discarded. This information would then be given back to the Profile Manager to update the Master Document with the higher-credibility and more structured information.

The ability to verify facts in a knowledge graph allows us to provide positive feedback to other modules which gives the system a few interesting characteristics that help immensely in its performance. Firstly, we can keep track of the sources of each statement of fact and using competitive learning algorithms, score the sources we are encountering as a function of how factually accurate their information is. This algorithm would then help in biasing the *Web Crawler* towards more credible sources and away from less credible ones. Secondly, we can perform relational machine learning on the knowledge graph to find which facts not present in the knowledge graph are most likely to be true, pass this information to the *Query Formulator*, and direct the system to investigate the fact.

Chapter 4

Results

The results produced at the end of each architecture component have been described in the result section. The result section also contains the progress and attempts with various modules in the architecture.

1. **Web Crawling Framework:** We managed to get sources and raw text output of company information from the web crawling framework.

2. **Information Classification Aggregation:** We managed to get a master document called **Company Profile** for each company that contains both sources and texts after classification for all the relevant company information and search results.

3. **Computational Fact-Checking using Knowledge Graph:** progress and attempts.

4.1 Web Crawling Framework

The purpose of the web crawling framework is to automatically navigate through the internet, get the text information we want on web pages and effectively store sources and raw text data in order for us to retrieve information. The functionality of the framework is achieved by completion of individual modules in it: the *Query Formulator*, the *Web Crawler*, the *Parser* and the *Web Resource Manager*. In practice, our goal is to get relevant company information. In Figure 4.1, we illustrate the original web pages and the final data outputs we get by implementing the web crawling framework.

We managed to build a database that contains sources and textual data of:

1. company filings including 10-K, 8-K and EX-21 for 52,629 companies,

2. TRI Facility reports and chemical usage sheets for 21,202 facilities,

3. Wikipedia infobox and page contents for 52,000+ companies,

4. product and ingredient information for 4,535 companies in EWG database,

5. subsidiary information returned by Google for 501 companies provided by Praedicat,Inc..

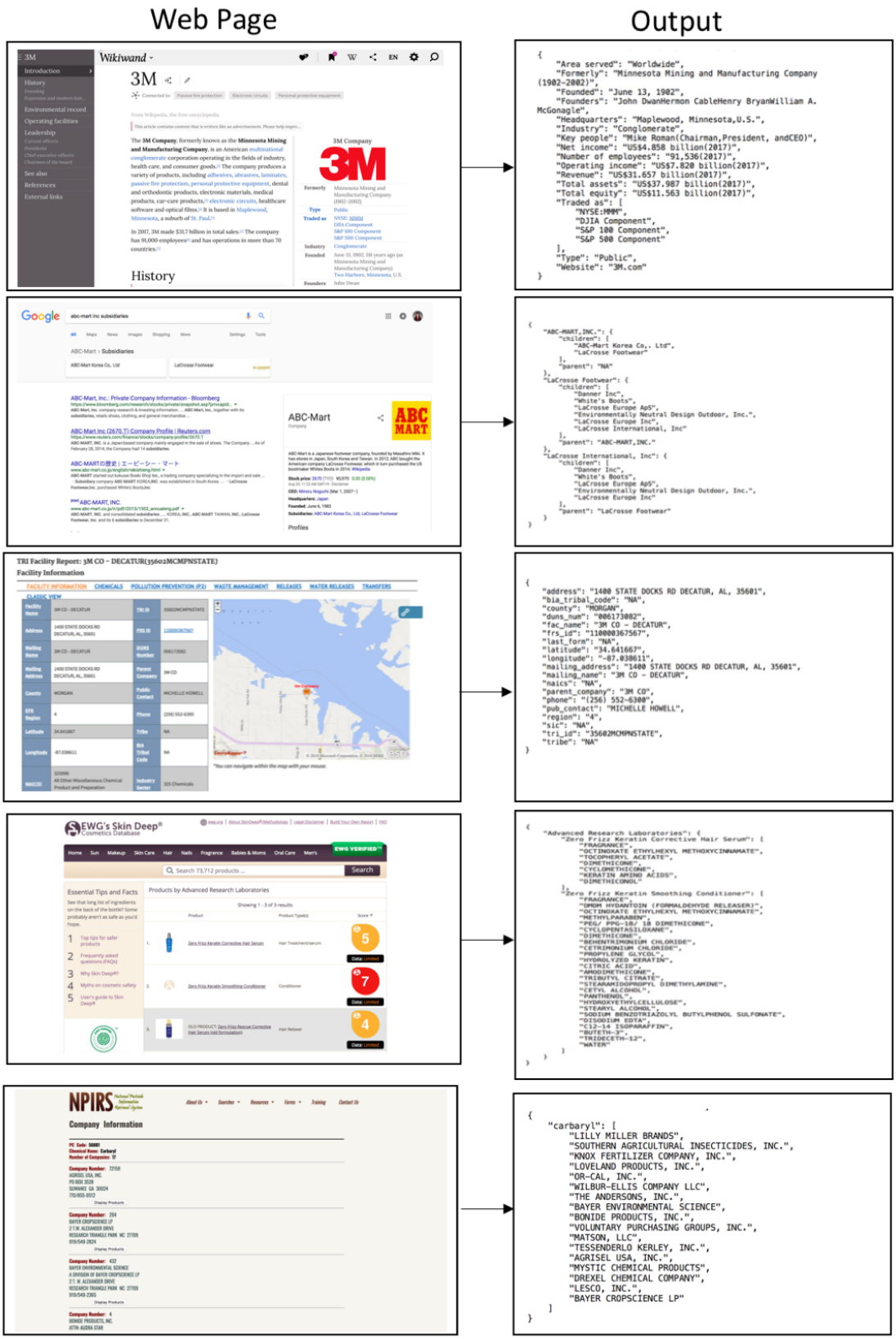


Figure 4.1: Web Pages and Web Crawling Framework Output

4.2 Information Classification & Aggregation

As discussed in Section 3.2.1, we tested two classifiers using the same method based on self-supervised learning: *Web Resource Doc2Vec* which classified on raw web-based data and only used "bad" tags and *Profile Manager Doc2Vec* which classified on financial statements using "good" and "bad" tags. All of the information was sentence and word tokenized before being passed to the classifier, regular expressions where applied to remove all non-alpha-numeric characters except apostrophes (') and hyphens (-) (we felt they affected the semantic meaning of words, but other symbols were exogenous), and all words were stemmed and lemmatized.

Our classifiers were both compared against Term Frequency-Inverse Document Frequency (TF-IDF) to provide a benchmark. The top two-hundred and fifty sentences of each classifier were selected and hand-classified into four categories: *Useful without Context*, *Useful with Context*, *Uninformative*, and *Incoherent* with the following descriptions:

Useful without Context	The statement provides insight into the company (management, practices, etc.) without needing context from the source document
Useful with Context	The statement provides insight into the company (management, practices, etc.) with context from the source document
Uninformative	The statement does not contain any useful information
Incoherent	The information does not have any meaningful interpretation

Classifiers are considered better if they are able to produce more sentences in the *Useful without Context* and *Useful with Context* categories, with the former being preferred.

(TF-IDF Results and Word2vec semi-supervised keyword generation to be added.)

The heuristics we used to label the training examples were chosen because we anecdotally noticed them to be indicative of "junk" text, but it could be the case that our choices led the model astray. Given more time or a pre-labeled data set of "good" information and "bad" Internet text, we could more than likely improve these heuristics and thus the overall approach. We would have also likely to have explored cosine similarity in different subspaces of the vector space. It is likely that some of the features in the feature vector representation of the words were not indicative for the type of classification we were trying to perform, and calculating cosine similarity on the optimal subspace of the feature space would have improved results.

The Natural Language Processing techniques (tokenizing, regular expressions, stemming, and lemmatization) we used to process the sentences before passing them to the sentence could also have contributed to the number of incoherent and uninformative sentences we produced, and given more time we would have established a Python dictionary mapping the raw text sentence to the unique document ID thus allowing us to map the ranked outputs back to the raw text sentences.

Using the information that was filtered using these heuristics and classifiers were ultimately fed into the *Profile Manager* and converted to Master Documents for analysts. The Master Documents are HTML files containing Wikipedia information, subsidiaries found using our Google subsidiary parser, the financial statements from the SEC parsers, any information in the *Profile Manager's* profile for the company, and the *Web Crawling Framework's* output with i-frames on one side and the filtered text on the other. The information contained is already filtered for relevance and has been aggregated onto one page, meaning it is higher value and easier to search. We produced thousands of documents for Praedicat based on lists we were given to stress test the system and the architecture is capable of produce as many as needed.

4.3 Computational Fact-Checking using Knowledge Graph

(Add results collected on text triple construction, using REVERB)

We discussed in Section 3.3 about various state-of-the-art algorithms in computational fact-checking on knowledge graphs. We were able to develop code for the proposed fact-checking algorithm, *StreamMiner*. Currently, we are still performing experiments on the proposed algorithm. Further, we also tested other state-of-the-art algorithms on numerous datasets and compared them based on their AUROC (Area under Receiver Operating Characteristic) Curve. The following tables illustrate the results:

Table 4.1: Fact-Checking Performance (AUROC) on Synthetic Datasets. Best Scores for each dataset are shown in bold.

Method	NYT-Bestseller	NBA-Team	Oscars	CEO	US-War	US-V.President	FLOTUS	Capital#2	Avg (S.E.)
KL-REL	96.32	99.94	97.67	**89.88**	86.34	87.29	98.32	**100.00**	94.27 (2.0)
KS	89.72	**99.96**	95.00	81.19	72.11	77.80	98.05	**100.00**	89.23 (3.9)
PredPath	**99.80**	92.31	99.97	88.67	**99.51**	**94.40**	**100.00**	99.68	**96.79 (1.6)**
KL	94.99	99.94	97.56	89.77	63.55	74.62	98.59	99.42	89.80 (4.8)

Table 4.2: Fact-Checking Performance (AUROC) on Real-World Datasets. Best Scores for each dataset are shown in bold.

Method	GREC Birthplace	GREC Deathplace	GREC Education	GREC Institution	WSDM Nationality	WSDM Profession	Avg (S.E.)
KL-REL	**92.54**	**90.91**	86.44	85.64	96.92	97.32	**91.63 (2.0)**
KS	72.92	80.02	**89.03**	78.62	**97.92**	**98.66**	86.20 (4.4)
PredPath	84.64	76.54	83.21	80.14	95.20	92.71	85.41 (2.9)
KL	92.10	90.49	62.32	**87.61**	96.05	91.36	86.65 (5.0)

The algorithms, KS [20], KL-REL [20], PredPath [18] and KL [2] were tested on synthetic and real-world datasets. More information about the datasets can be found on [20]. *PredPath* showed the average best performance on Synthetic datasets (Table: 4.1) and *KL-REL* performed the best on the real-world datasets (Table: 4.2). These results paved our motivation towards combining the specificity measure (Eq: 3.4) and path mining into the *StreamMiner* algorithm.

Chapter 5

Conclusion & Future Direction

5.1 Conclusion

The contributions in this paper are threefold: a web crawling, classification, and aggregation architecture which is able to accumulate information at scale that is able to provide useful Information Extraction capabilities to companies like Praedicat, Inc., a classification technique for Information Extraction, and a proposed architecture which would allow for checking the factual correctness and credibility of the information found as well as direct the system's search.

The Architecture laid in Sections 3.1 and 3.2 were fully implemented and can be found at `https://github.com/himahuja/pcatxcore`, with documentation for the implementation available there or in Appendix D. It was stress-tested over several weeks using data graciously provided by Praedicat, Inc. and ultimately provided gigabytes of useful data for the analyst team.

Our classification technique achieved the following: (Still to be added). Further, the novel computational fact-checking algorithm, *StreamMiner* is in its rudimentary testing phase and will provide the whole architecture the capability of fact-checking of the retrieved information.

5.2 Future Direction

Due to the time constraints on our work with the Research in Industrial Projects for Students program, we were unable to fully implement the entire architecture proposed, chiefly the portion discussed in Section 3.3. The section discusses how we approached conversion of raw text to subject-predicate-object triples, and our proposed computational fact-checking algorithm which is an extension on the existing *PredPath* algorithm. The techniques we planned to implement for the positive feedback discussed in Section 3.3 were entirely based on Relational Machine Learning for Knowledge Graphs and an overview is present in our Literature Review (Section 2.2.1).

Appendix A

Glossary

Anchored Predicate Path:	a directed, typed sequence of edges with typed-endpoints $P^k = o_1 \xrightarrow{p_1} \xrightarrow{p_2} ... \xrightarrow{p_{k-1}} o_k$ [18].
Adjacency Matrix:	an n n matrix whose element A ij equals 1 if there is an edge joining vertices i and j, otherwise it is zero [5].
Complete Graph:	a graph in which every two vertices are adjacent.
Connected Component:	maximal connected subgraph of a graph [5].
Graph:	a pair of sets (**V**, **E**), where **V** is a set of vertices or nodes and **E** is a subset of $\mathbf{V}^2$, the set of unordered pairs of elements of V [5].
Homophily:	the tendency of individuals to associate and bond with others who are similar [15].
Information extraction (IE):	the task of automatically extracting structured information from unstructured and/or semi-structured machine-readable documents [1].
Knowledge Graph:	**G** is an ordered pair **G** = (**V**, **E**) where **V** is a set of concept nodes and **E** is a set of predicate edges [2].
Line Graph:	graph whose vertices are the edges of the original graph; vertices of the line graph are linked if the corresponding edges in the original graph are adjacent [5].

Ontology:	a set of concepts and categories in a subject area that shows their properties and the relations between them [19].
Path:	a sequence of vertices $\{v_1, ..., v_n\}$ such that for all v_i, $i \neq 1$, v_{i-1} and v_i are adjacent.
Resource Description Framework (RDF):	a World Wide Web Consortium standard for publishing interlinked data on the web [19].
Semantic Networks:	*See Knowledge Graph.*
Subject-Predicate-Object Triple:	an ordered triple where the subject and object are concepts and the predicate describes the relationship between the two concepts [2].
Transitive:	whenever an a relates to b and b relates to c then a relates to c
Vertex (Node):	A node v is a terminal point or an intersection point of a graph. It is the abstraction of a location such as a city, an administrative division, a road intersection or a transport terminal (stations, terminuses, harbors and airports). [16]
Edge (Link):	An edge e is a link between two nodes. The link (i, j) is of initial extremity i and of terminal extremity j. A link is the abstraction of a transport infrastructure supporting movements between nodes. It has a direction that is commonly represented as an arrow. When an arrow is not used, it is assumed the link is bi-directional [16].

Appendix B

Abbreviations

API - Application Programming Interface

CIK - Central Index Key

EPA - Environmental Protection Agency

HTML - Hypertext Markup Language

JSON - JavaScript Object Notation

NAICS - North American Industry Classification System

PDF - Portable Document Format

RDF - Resource Description Framework

RDFS - Resource Description Framework Schema

SEC - Securities and Exchange Commission

SIC - Standard Industrial Classification

TRI - Toxics Release Inventory

URI - Uniform Resource Identifier

KG - Knowledge Graph

URL - Uniform Resource Locator

UUID - Universally Unique Identifier

AUROC - Area under Receiver Operating Characteristic

Appendix C

Graph Theory

Introduction

We used techniques and ideas that stem from Graph Theory extensively in our work. Here we briefly cover some basic topics and ideas we discuss in our work from the large field of Graph Theory. Graphs studied in Graph Theory have their specificity and are thus different from general geometric graphs. We refer to parts of such graphs as **vertices** or **nodes** which are connected by **edges** or **links** as illustrated in Figure C.1 below [16].

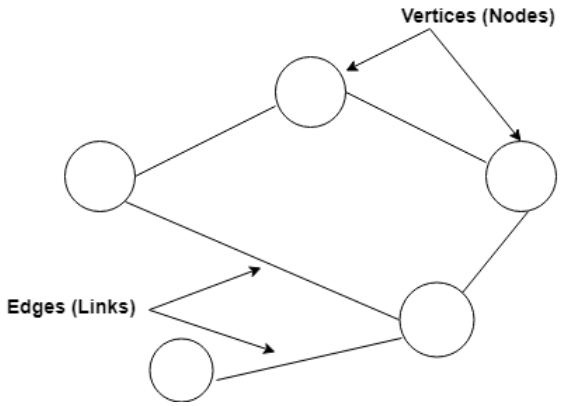


Figure C.1: Vertices and Edges of a Graph

Connectivity And Transitivity Of A Graph

In the study of knowledge graph for computational fact-checking, we care about the **connectivity** and **transitivity** of a graph. A graph is **connected** if all vertices are linked by edges [16]. If not all vertices are linked by edges, the graph is disconnected. Some of its vertices linked by edges form a **connected component**, which is also called a subgraph as illustrated in Figure C.2 below. **Transitivity**, on the other hand, refers to the extent that a relation between connected vertices is transitive [14]. An example of perfect transitivity is when we say an entity x is related to another entity z to the same extent as entity y is related to z because x is related to y and y is related to z in a knowledge graph. The entities x, y, z are thus connected to each other and form a connected component. Perfect transitivity is very rare in general and we do not want relations to be perfectly transitive

to each other which will result in a large number of incorrect relations.

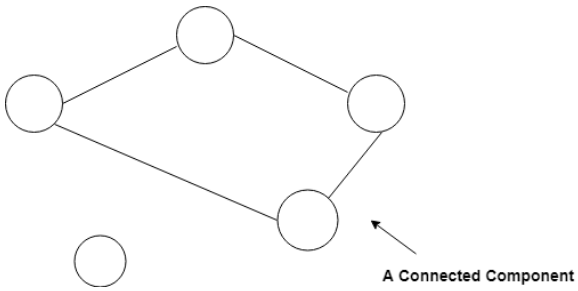


Figure C.2: A Connected Component

Line Graph And Adjacency Matrix

We can represent a dense graph by an **adjacency matrix** [16]. An adjacency matrix is a square matrix $A = A(D)$ in which the entries can be either 1 or 0. The entry is 1 in the i, j cell when there is a path from the ith point to the jth point and it is 0 otherwise [6]. The approach known as Relational Knowledge Linker that is used to capture the semantic similarity of the relations along the path in a knowledge graph that we have discussed represents the **line graph** of the knowledge graph by an adjacency matrix. In Graph Theory, the idea of a line graph was first proposed by Harary Frank in his work [7]. The line graph $L(G)$ of a graph G is obtained by drawing a vertex for each edge in G and connecting two vertices if and only if the edges in G share a common vertice as illustrated in Figure C.3 below [7].

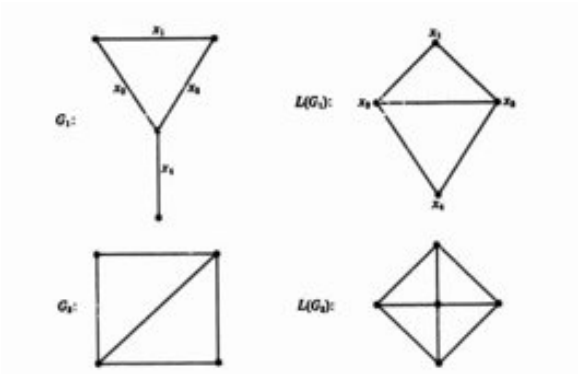


Figure C.3: Line Graph [7]

Appendix D

Documentation

D.0.1 PCATParser

eightk_parser(link)

Parses an SEC document known as an 8-K

Parameters

- link (string) : the URL for the 8-K

Returns

- string : the important text for the 8-K

ex21_parser(link)

Parses an SEC document known as an EX-21

Parameters

- link (string) : the URL for the EX-21

Returns

- list of strings : the subsidiaries in the company listed in the EX-21

get_PDF_content(query_string, link

Gets all of the text from a PDF document

Parameters

- query_string (string) : the query that generated the PDF document
- link (string) : the URL for the document

Returns

- string : the visible text in the PDF

parser_iter(query_string, linkList)

Parses the URLs in linkList using a timeout of 60 seconds on each page (a la try_one) and yields them as dictionaries.

Parameters

- query_string (string) : the generating query, default is "test"
- linkList (list of strings) : list of URLs for the documents you would like to parse

Returns

- dict (yields many)

  - dict['text'] (string) : the visible text on the web page
  - dict['html'] (bytes) : the HTML code of the page (if it is HTML based)
  - dict['pdf'] (bytes) : the PDF code of the page (if it is PDF based)

parse_single_page(link, query_string = "test")

Gets all of the text from web page

Parameters

- link (string) : the URL for the document
- query_string (string) : the generating query, default is "test"

Returns

- tuple (bytes, string) : the source code (HTML/PDF) of the web page and the visible text

tag_visible(element)

Determines if an HTML tag is visible

Parameters

- element (BeautifulSoup.element) : an HTML element

Returns

- bool : True if the element is visible, False else

tenk_parser(link)

Parses an SEC document known as an 10-K

Parameters

- link (string) : the URL for the 10-K

Returns

- string : the important information in the 10-K

text_from_html(body)

Gets all of the visible text from the body of an HTML document

Parameters

- body (string) : the body of an HTML document

Returns

- string : the visible text in the body

try_one(func, t, **kwargs)

Calls the function with the keyword arguments and after t seconds, interupts the call and moves on

Parameters

- func (function) : the function to be called
- t (int) : the number of seconds
- **kwargs (keyword-arguments) : arguments you'd like to pass to func

Returns

- func's return type

wikiParser(company)

Search the Wikipedia page for a company and get wikipedia infobox together with all other contents

Parameters

- company (str) : the company you would like to query Wikipedia for

Returns

- tuple
  - dict : a dictionary of all other contents on wikipedia
  - dict : a dictionary of wikipedia infobox
  - str : page title
  - str : page url
  - beautifulsoup.table : wikipedia infobox HTML

D.0.2 ProfileManager

__init__(rel_path=None)

Sets the rel_path variable and reads in various mappings including:

- CIK (Central Index Key) to name
- Name to CIK (Central Index Key)
- Name to (a list of) aliases
- NAICS (North American Industry Classification System) to description of classification
- SIC (Standard Industrial Classification) to description of Classification
- NAICS to SIC
- SIC to NAICS

Parameters

- rel_path (string) : the relative path from the script to the parent folder of where your data will be housed. ProfileManager always assumes that data will be held in "data/profilemanager/data" and profiles will be held in "data/profilemanager/profiles" so this allows you to orient your ProfileManager instance to your data source.

Returns

- None

__contains__(key)

Returns true if the key is found in the ProfileManager instance. Checks the list of CIK numbers, names, and aliases for matches.

Parameters

- key (string) : can be a CIK (Central Index Key) code, name, or alias

Returns

- None

__getitem__ and get(key)

Gets profile identified by the key

Parameters

- key (string) : can be a CIK (Central Index Key) code, name, or alias

Returns

- dict : A dictionary which is the profile if found, else None

__iter__(instances=1, iam = 0)

A generator that yields the profiles of the contained corporate entities with the ability to be accessed by multiple instances at once.

Parameters

- instances (int) : the number of instances using the iterator (default = 1)
- iam (int) : the current instance's assignment [0-*instances*) (default = 0)

Returns

- dict : A dictionary which is the profile if found, else None

__len__()

Returns the number of CIK (Central Index Key) codes in the instance.

Returns

- int : number of CIK (Central Index Key) codes in the instance

__repr__() and __str__()

Returns a sorted and indented dictionary of CIK (Central Index Key) codes to names of the profiles contained.

Returns

- str : a sorted and indented representation of the CIK (Central Index Key) to names map

build_aliases()

Builds an internal list of aliases from the contained items' names and aliases fields.

Returns

- None

cik_to_alias(cik)

Returns a list of aliases of the business entity identified by the CIK (Central Index Key) code.

Parameters

- cik (string) : is the CIK (Central Index Key) code of a business

Returns

- list of strings : a list of aliases associated with the CIK code

cik_to_description(cik)

Returns a list of descriptions of the industries of the NAICS (North American Industry Classification System) and SIC (Standard Industrial Classification) codes of the business entity identified by the CIK (Central Index Key) code.

Parameters

- cik (string) : is the CIK (Central Index Key) code of a business

Returns

- list of strings : a list of descriptions of business activities associated

cik_to_naics(cik)

Returns the NAICS (North American Industry Classification System) codes of the business entity identified by the CIK (Central Index Key) code.

Parameters

- cik (string) : is the CIK (Central Index Key) code of a business

Returns

- list of strings : the NAICS codes associated with the CIK code

cik_to_name(cik)

Returns the name of the business entity identified by the CIK (CentralIndex Key) code.

Parameters

- cik (string) : is the CIK (Central Index Key) code of a business

Returns

- string : the name associated with the CIK code

cik_to_sic(cik)

Returns the SIC (Standard Industrial Classification) codes of thebusiness entity identified by the CIK (Central Index Key) code.

Parameters

- cik (string) : is the CIK (Central Index Key) code of a business

Returns

- list of strings : the SIC codes associated with the CIK code

clean_financial_statements()

Removes the text field of all ten_k's, eight_k's, and EX21's which are either the empty string or None.

Returns

- None

generate_profiles()

The code uses the various mappings to aggregate the information into one profile for each business entity.

Currently the code uses two additional JSON files to generate profiles:

- a map from CIK (Central Index Key) to SIC (Standard Industrial Classification)
- a map from SIC (Standard Industrial Classification) to NAICS (North American Industrial Classification System)

Returns

- None

get_aliases()

A getter function for the the list of aliases

Returns

- list of strings : a list of names and aliases of entities in the instance

get_docs_by_sentence(instances,iam)

An generator function of the sentences of the documents contained withthe ability to be accessed by multiple instances at once in a safe way.

Parameters

- instances (int) : the number of instances using the iterator (default = 1)
- iam (int) : the current instance's assignment [0-*instances*) (default = 0)

Returns

- list of tuples (string, string) : A list tuples representing the sentences of the documents contained and the IDs of the sentences (Yields)

get_resources_by_company(item)

A getter for the documents and associated URLs of the resources by company

Parameters

- item (dict) : a profile

Returns

- list of tuples (string, string) : A list tuples representing the text of the documents contained and the URLs of the documents

get_texts()

A getter for the documents contained

Returns

- list of tuples (string, string) : A list tuples representing the text of the documents contained and the IDs of the documents (Yields)

naics_to_description(naics)

Returns a list of descriptions of the industrial code.

Parameters

- naics (string) : is a NAICS (North American Industry Classification System) code.

Returns

- list of strings : a list of descriptions of the industrial code

parse_sec_docs(filename)

The code parses the filings which haven't already been parsed, iterating on the CIK codes contained in filename. This means if the CIK codes are disjoint, this method can safely be run in parallel.

Parameters

- filename (string) : the name of a JSON file in "data/profilemanager/data/edgardata/JSON". It must be a dictionary from CIK (Central Index Key) to a dictionary containing the keys "10K", "8K", and "EX21". These keys must map to a list of dictionaries each containing the keys "time_of_filing" and "url".

Returns

- None

parse_wikipedia(parse_list)

Gets the information on the Wikipedia pages for the companies in parse_list

Iterates on the companies contained and searching Wikipedia for the name field, then saves the returned page's parsed table and text in dictionaries ("wiki_table" and "wiki_page" respectively). The table is a dictionary from heading to values and the page is a dictionary from section headings to content.

Parameters

- parse_list (list of dicts) : list of profiles which you would like to get the Wikipedia information for

Returns

- None

save_aliases()

Saves the aliases list to "profilemanager/data/aliases.json" using rel_path

Returns

- None

update_profile(profile)

Writes the current instance of the profile to the JSON

Parameters

- profile (dict) : profiles which you would like update the saved version of

Returns

- None

write_to_raw_text()

Writes all of the contained documents to "profilemanager/raw_text"

Returns

- None

D.0.3 Site_Crawler_Parser_All

company_to_product(company, driver)

Search a company name on EWG and get all products made by the company in EWG database

Parameters

- company (str) : A company name to find products for
- driver (selenium.webdriver.Chrome) : Chrome driver after calling 'driver = setDriver()'

Returns

- dict
  - COMPANY (str) : a list of products made by the company

get_comp_name(text)

Extract relevant content of company name in a html tag

Parameters

- text (str) : raw content in a html tag

Returns

- str : a clean company name after junk texts are filtered

get_parent_child_dict(company,parent,children_list)

Build a dictionary that contains parent company, subsidiary company information for a certain company

Parameters

- company (str) : company name to build dictionary for
- parent (str) : The parent company name for the company
- children_list (list of strings) : list of subsidiary names of the company

Returns

- dict :
  - parent (str) : the parent company name
  - child (list of str) : a list of subsidiary names of the company

get_recursive_sub(company, driver)

Search "COMPANY_NAME+subsidiaries" RECURSIVELY on Google chromedrivers directory to get all-level subsidiaries of a company and build a master dictionary that contains all-level subsidiary information for a company

Parameters

- company (str) : company name to find subsidiary for
- driver (selenium.webdriver.Chrome) : Chrome driver after calling 'driver = setDriver()'

Returns

- dict
  - company (str) :
    - parent (str) : the parent company of the company,'NA' if not found
    - child(list): a list of subsidiary names

get_tri_dict(tri_id, driver)

Open facility report page and scrape facility information into a dictionary

Parameters

- tri_id (str) : TRI facility id used as a unique identifier for a facility on TRI Search
- driver (selenium.webdriver.Chrome) : Chrome driver after calling 'driver = setDriver()'

Returns

- fac_dict (dict)
  - fac_name(str): Facility Name
  - tri_id(str): TRI facility ID
  - address(str): Facility Address
  - frs_id(str): FRS ID
  - mailing_name(str): Facility Mailing Name
  - mailing_address(str): Facility Mailing Address
  - duns_num(str): Facility Duns Number
  - parent_company(str): Facility's Parent Company Name
  - county(str): County
  - pub_contact(str): Public Contact Name
  - region(str): EPA Region Code
  - phone(str): Contact Number
  - latitude(str): Latitude
  - tribe(str): Tribe
  - longitude(str): Longitude
  - bia_tribal_code(str): BIA Tribal Code
  - naics(str): Naics Code
  - sic(str): SIC Code
  - last_form(str): Last Year of Report

google_sub(company, driver)

Search "COMPANY_NAME+subsidiaries" on Google chromedrivers directory and scrape the knowledge graph results of subsidiary names returned by Google on the top

Parameters

- company (str) : A company name to find subsidiary for
- driver (selenium.webdriver.Chrome) : Chrome driver after calling 'driver = setDriver()'

Returns

- list : a list of subsidiary names(str)

hazard_to_company(chemical,driver)

Search NPIRS by entering a chemical name and get a list of companies that use the chemical in their products in NPIRS database

Parameters

- chemical (str) : a hazard name
- driver (selenium.webdriver.Chrome) : Chrome driver after calling 'driver = setDriver()'

Returns

- list of strings : a list of companies that use the hazard

product_to_ingredient(comp_prod_dict,driver)

Search a product name on EWG, get all ingredients in the product in EWG database, and build a master dictionary that contains information for company-products-ingredients

Parameters

- comp_prod_dict (dict) : dictionary that contains company to products information after calling 'comp_prod_dict = company_to_product(company, driver)'
- driver (selenium.webdriver.Chrome) : Chrome driver after calling 'driver = setDriver()'

Returns

- dict
  - COMPANY (str) :
    - PRODUCT (str): a list of ingredients in the company product

remove_null(comp_list)

Remove null values in a company list

Parameters

- comp_list (list of strings) : a list of companies

Returns

- str : a clean list of company names with no null values

setDriver(headless = False)

Sets a selenium webdriver object for running web-crawlers on various systems. Note: Requires chromedrivers for various platforms in a chromedrivers directory

Parameters

- headless (bool) : if True, sets a headless browser. if False (Default), sets a browser with head

Returns

- selenium.webdriver.Chrome : driver with standard option settings

wikiParser(company)

Search the Wikipedia page for a company and get wikipedia infobox together with all other contents

Parameters

- company (str) : the company you would like to query Wikipedia for

Returns

- tuple
  - dict : a dictionary of all other contents on wikipedia
  - dict : a dictionary of wikipedia infobox
  - str : page title
  - str : page url
  - beautifulsoup.table : wikipedia infobox HTML

D.0.4 webcrawlAll

crawlerWrapper(search_query, enging, doSetDriver, headless = False)
Takes in the query to search for on a portal. NOTE: Saves to file, does not return anything. Currently supported portals:

1. google: searches the google page for entered company
2. sec10k: searches the 10k filing for that company
3. sec10kall: finds the 10Ks of all the companies
4. secsic10k: uses the SIC codes to gather all the companies' 10K in all SICs
5. generalSEC: performs any general query on SEC using `urlmaker_sec`
6. sitespecific: Performs Crawling specifically on any particular website
7. tri: Returns the TRI page for given facility ID
8. everything-all: finds the 8Ks, 10Ks and EX-21s of all the companies on the SEC website, via CIK

Parameters

- search_query (dict) : the format for search query for different engines is as follows:
  1. google
  - name (str): the mandatory portion of the search query
  - aliases (str[]): optional words of the search query
  - filetype (str): filetype to be searched for
  2. sec10k:
  - cik (str): String typed CIK code of the company
  - dateStart (str): Starting date of filings, '/' seperated date, MM/DD/YYYY
  - dateEnd (str): Ending date of filings, '/' seperated date, MM/DD/YYYY
  3. sec10kall:
  - None
  4. secsic10k:
  - None
  5. generalSEC:
  - searchText (str) : Company name to be searched (Default: '*')
  - formType (str): Type of the document to be retrieved (Default: '1')
  - sic (str): SIC code for the companies to be searched (Default: '*')
  - cik (str): CIK code for the company to be searched (Default: '*')
  - startDate (str): Start date of the produced results (YYYYMMDD) (Default: '*')
  - endDate (str): End date of the produced results (YYYYMMDD) (Default: '*')
  - sortOrder (str): Ascending (Value = 'Date') or Descending (Value = 'Reverse-Date') retrieval of results, (Default: 'Date')
  6. tri:

- tri_id (str): TRI ID of the facility

7. google-subs:

- name: name of the company whose subsidiaries have to be discovered

8. everything-all:

- None

- search_query (str)
  - Default Settings:
    - -p0: only parse URLS, don't download anything
    - -%I : make an index of links
    - set depth of 5
    - language preference: en
    - -n: get non-HTML files near an HTML
  - Parameters
    - search_query['name'] (url of the website we need to download)
    - -O output directory
    - -r set the depth limit
    - -m, non-HTML,HTML file size limit in bytes
    - %e, number of external links from the targetted website
    - '%P0' don't attempt to parse link in Javascript or in unknown tags
    - -n get non-HTML files near an HTML-files (images on web-pages)
    - t test all URLs
    - -%L , loads all the links to be tracked by the function
    - K0 Keep relative links
    - K keep original links
    - -%l "en, fr, *" language preferences for the documents
    - -Z debug log
    - -v verbose screen mode
    - I make an index
    - %I make a searchable index
    - -pN priority mode (0): just scan (1): just get HTML (2): just get non-HTML (3): save all files (7): get HTML files first, then treat other files
- engine (str) : specify which type of crawler to use, refer module summary for options

Returns

- None

linkFilter_google(url)

Filters out the links of social media websites from the returned google search results using `filterList` defined implicitly.

Parameters

- url (str) : URL to be tested against `filterList`

Returns

- int : returns 0 (is a social media link), 1 (is not a social media link)

search_google(query, driver, number_of_pages)

Searches Google websites for the top page results

Parameters

- query (dict)
  - name (str): the mandatory portion of the search query
  - aliases (str[]): optional words of the search query
  - filetype (str): filetype to be searched for
- driver (selenium.webdriver.Chrome) : An instance of browser driving engine
- number_of_pages (int) : number of pages of Google web results

Returns

- list of strings : list of links returned from the Google search engine

setDriver(headless = True)

Sets a selenium webdriver object for running web-crawlers on various systems. Note: Requires chromedrivers for various platforms in a chromedrivers directory

Parameters

- headless (bool) : if True, sets a headless browser. if False (Default), sets a browser with head

Returns

- selenium.webdriver.Chrome : driver with standard option settings

urlmaker_sec(queryDic)

Produces the URL, which can be entered into the search (Designed for SEC.gov)

Parameters

- queryDic (dict)
  - searchText (str): Company name to be searched (Default: '*')
  - formType (str): Type of the document to be retrieved (Default: '1')
  - sic (str): SIC code for the companies to be searched (Default: '*')
  - cik (str): CIK code for the company to be searched (Default: '*')
  - startDate (str): Start date of the produced results (YYYYMMDD) (Default: '*')
  - endDate (str): End date of the produced results (YYYYMMDD) (Default: *')
  - sortOrder (str): Ascending (Value = 'Date') or Descending (Value = 'Reverse-Date') retrieval of results, (Default: 'Date')

Returns

- str : URL to be searched on the SEC website

Bibliography Including Cited Works

[1] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni, *Open information extraction from the web*, in Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI'07, San Francisco, CA, USA, 2007, Morgan Kaufmann Publishers Inc., pp. 2670–2676.

[2] G. L. Ciampaglia, P. Shiralkar, L. M. Rocha, J. Bollen, F. Menczer, and A. Flammini, *Computational fact checking from knowledge networks*, PLOS ONE, 10 (2015).

[3] O. Etzioni, A. Fader, J. Christensen, S. Soderland, and M. Mausam, *Open information extraction: The second generation*, in Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One, IJCAI'11, AAAI Press, 2011, pp. 3–10.

[4] A. Fader, S. Soderland, and O. Etzioni, *Identifying relations for open information extraction*, in Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP '11, Stroudsburg, PA, USA, 2011, Association for Computational Linguistics, pp. 1535–1545.

[5] S. Fortunato, *Community detection in graphs*, Physics Reports, 486 (2010), pp. 75 – 174.

[6] F. Harary, *The determinant of the adjacency matrix of a graph*, SIAM Review, 4 (1962), pp. 202–210.

[7] ——, *Graph Theory*, Massachusetts: Addison-Wesley, 1972.

[8] Q. V. Le and T. Mikolov, *Distributed representations of sentences and documents*, CoRR, abs/1405.4053 (2014).

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Scoring, term weighting and the vector space model*, Introduction to information retrieval, 100 (2008), pp. 2–4.

[10] J. L. Martinez-Rodriguez, I. Lopez-Arevalo, and A. B. Rios-Alvarado, *Openie-based approach for knowledge graph construction from text*, Expert Systems with Applications, 113 (2018), pp. 339 – 355.

[11] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, CoRR, abs/1301.3781 (2013).

[12] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, CoRR, abs/1310.4546 (2013).

[13] T. Mikolov, W.-t. Yih, and G. Zweig, *Linguistic regularities in continuous space word representations*, in Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Atlanta, Georgia, June 2013, Association for Computational Linguistics, pp. 746–751.

[14] I. B. Mohammad Aghagolzadeh and H. Radha, *Transitivity matrix of social network graphs*.

[15] M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich, *A review of relational machine learning for knowledge graphs: From multi-relational link prediction to automated knowledge graph construction.*, CoRR, abs/1503.00759 (2015).

[16] Satish, *Introduction to graph theory*, International Journal of Research in Engineering and Applied Sciences, 3, pp. 35–43.

[17] C. Shao, P.-M. Hui, L. Wang, X. Jiang, A. Flammini, F. Menczer, and G. L. Ciampaglia, *Anatomy of an online misinformation network*, PLoS ONE, 13 (2018), p. e0196087.

[18] B. Shi and T. Weninger, *Fact checking in large knowledge graphs - A discriminative predicate path mining approach*, CoRR, abs/1510.05911 (2015).

[19] P. Shiralkar, *Computational fact checking by mining knowledge graphs*, 2017.

[20] P. Shiralkar, A. Flammini, F. Menczer, and G. L. Ciampaglia, *Finding streams in knowledge graphs to support fact checking*, CoRR, abs/1708.07239 (2017).

[21] T. Simas and L. M. Rocha, *Distance closures on complex networks*, CoRR, abs/1312.2459 (2013).

[22] S. Van Dongen and A. J. Enright, *Metric distances derived from cosine similarity and pearson and spearman correlations*, arXiv preprint arXiv:1208.3145, (2012).

[23] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu, *Toward computational fact-checking*, Proc. VLDB Endow., 7 (2014), pp. 589–600.