Supporting Data Analytics Applications Which Utilize Cognitive Services

Arun Iyengar
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
Email: aruni@us.ibm.com

***Abstract*—A wide variety of services are available over the Web which can dramatically improve the functionality of applications. These services include information retrieval (including data lookups from a variety of sources and Web searches), natural language understanding, visual recognition, and data storage. A key problem is how to provide support for applications which use these services. This paper presents a rich software development kit (SDK) which accesses these services and provides a variety of features applications need to use these services, optimize performance, and compare them. A key aspect of our SDK is its support for natural language understanding services. We also present a personalized knowledge base built on top of our rich SDK that uses publically available data sources as well as private information. The knowledge base supports data analysis and reasoning over data.**

1. Introduction

A wide variety of services are currently available over the Web. These include search engines as well as information retrieval services which provide data from data repositories such as DBpedia [1], [2], Wikidata [3], [4], and Yago [5]. Other Web services provide biomedical, financial, economic, geographical, climate, and other types of data. Cognitive services provide natural language processing, speech recognition, and video recognition. There are also services which provide data transformations from one format to another as well as data extraction. Sophisticated and powerful applications can be built using these services.

Software development kits (SDKs) exist to access a variety of cloud services such as those from Amazon [6], IBM [7], and Microsoft [8]. Cloud services typically expose HTTP interfaces and return data in a standard format such as JSON or XML. SDKs make it easier to access cloud services by encapsulating HTTP calls to cloud services within method calls of a programming language such as Java, Python, or Node.js (which is actually a JavaScript runtime built on Chrome's V8 JavaScript engine). There are often different SDKs to accommodate the different programming languages application writers are likely to use for applications which access cloud services. SDKs for cloud services generally only offer basic features for accessing cloud services such as logging into an account to allow access to a particular set of services and wrapping HTTP calls to the services in a function, method, or procedure which makes it easier to invoke the HTTP calls. There is a clear need for additional features which provide considerably more support to applications invoking cloud services.

We have developed a *rich SDK* which improves upon previous SDKs by providing a much broader set of features for supporting applications accessing services (Figure 1). This paper describes the key features of our rich SDK. These features include the following:

- Our rich SDK can collect data on services related to performance, availability, and the quality and accuracy of responses.
- Multiple services providing similar functionality may exist. Our rich SDK can rank services and help an application select the most appropriate service (s) to invoke. The rich SDK might use data it has collected monitoring services in order to rank and select the right ones to invoke.
- The rich SDK can handle service request failures and retry failed service requests. In some cases, another service might be invoked if the first service request fails.
- The rich SDK can cache data from remote services locally to improve performance and avoid the need to make redundant service calls. Caching can also help an application to continue executing if the application has poor connectivity with a service or the service is unresponsive.
- The rich SDK provides efficient methods to invoke services both synchronously and asynchronously.

A key use case for our rich SDK is to help applications use intelligent services which understand language and perform visual recognition. These services can add entire new functionality to applications. The rich SDK can be used to enhance one or more natural language (or visual) recognition services. Users can analyze several text documents and aggregate results from this analysis. The rich SDK combines Web searches with natural language processing and has features for analyzing results from natural language understanding services and storing these results persistently.

This paper also presents a personal knowledge base built on top of our rich SDK. The personal knowledge base allows data to be stored in multiple ways. We provide relational database management systems (RDBMS), key-value stores,

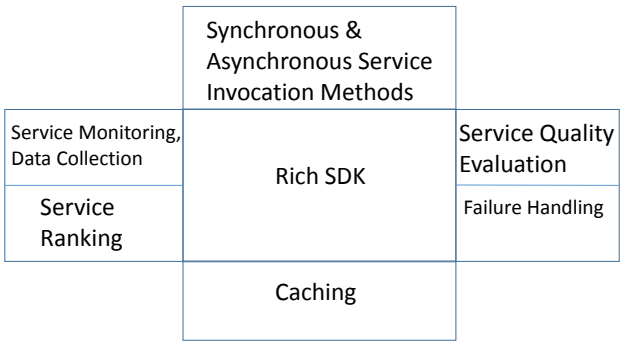

Figure 2: This figure depicts several key features provided by our rich SDK.

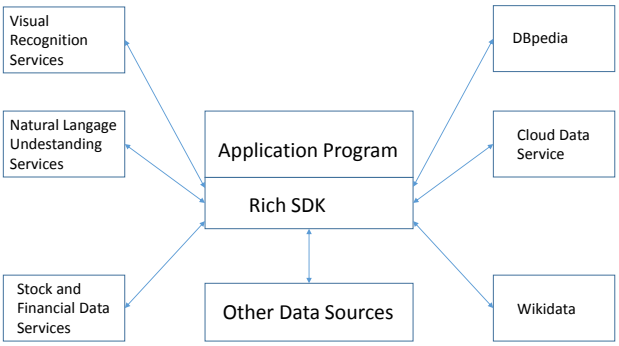

Figure 1: Our rich SDK provides interfaces and enhanced features for allowing applications to access a wide variety of services offered over the Web.

RDF [9] triple stores, and support for storing and retrieving data organized as comma-separated values (CSV) in files. The personal knowledge base can also convert data from one format to another. Users have access to a wide variety of external data sources via the rich SDK. It is also very easy for users to enter new facts into the personal knowledge base. The personal knowledge base can analyze data for patterns and perform predictive analytics; it also provides inferencing capabilities.

The remainder of the paper is structured in the following way. Section 2 describes our rich SDK and its key features. Section 3 describes our personalized knowledge base. Section 4 discusses related work. Finally, Section 5 concludes the paper.

2. Rich SDK Features

Our rich SDK provides several features which enhance the functionality of applications which use services offered over the Web. The features which our rich SDK provides go significantly beyond features provided by previous SDKs. Applications can use cloud services along with our rich SDK to dramatically increase functionality and performance. Figure 2 depicts key features of our rich SDK.

For example, a cognitive data analytics application may make several calls to services for operations (e. g. Web searches, language recognition, image recognition) that cannot be handled locally. These services may have costs associated with them. The cost may be both monetary as well as computational, such as latency to get a response from a service. In order to mitigate the latency, the rich SDK allows responses from services to be cached. That way, if a subsequent request is made for the same data, the data can be obtained from the cache which avoids the overhead for making a call to a remote service. Caching will not be applicable for all remote services. For example, if a remote service is performing a storage operation in a remote server, then the remote service call needs to take place. Furthermore, consistency issues may arise in which a cached value is obsolete.

Remote service costs can also be mitigated by having several candidate services which provide similar functionality. For example, there are a variety of storage services, search engines, and natural language understanding services offered by different service providers. Our rich SDK allows an application to choose from among multiple services. For example, we provide several different options for storing data. We also allow different search engines to be used. Candidate services may have different costs associated with them. Our rich SDK provides the ability to determine the latency for each of the candidates. That way, an application can pick a service which has the lowest expected latency. The rich SDK computes both average latencies and maintains histories of latencies allowing users to compare latency distributions. Users can also provide methods to rate the quality of different services.

Latency values can also be correlated with one or more parameters. For example, the behavior of a service may depend on the size of an argument passed to the service. The time for storing an object of size a will generally increase with a. The overhead as a function of size may increase differently for different services. Service s_1 may have the lowest latency for storing small objects, while s_2 may have the lowest latency for storing large objects. Our rich SDK can record the latency of services as a function of one or more *latency parameters*. Latency parameters are provided by users; an example of a typical latency parameter is the size of an argument passed to a service. The rich SDK can store past latency measurements along with the latency parameters resulting in each latency measurement. It can then predict the latency of a service invocation based on the latency parameters associated with the service invocation. This allows a data analytics application to select a service with the lowest expected latency based on the latency parameters.

In order to further mitigate the latency for accessing a remote service, the rich SDK provides asynchronous method calls to services. Asynchronous method calls execute in a separate thread or process from the main application and

allow the main application to continue executing without blocking while waiting for the service to return a response. In Java, asynchronous method calls can be implemented using futures. Java provides a Future interface which can be used for representing the result of an asynchronous computation. The Future interface provides methods to determine if the computation corresponding to a future has completed execution, to wait for the computation to complete if it has not finished executing, and to retrieve the result of the computation after it has finished executing. Our rich SDK implements asynchronous calls to services using the ListenableFuture interface [10]. The ListenableFuture interface extends the Future interface by giving users the ability to register callbacks which comprise code to be executed after the future completes execution.

Asynchronous service invocations are appropriate in situations in which the result from a service invocation is not needed immediately. For example, suppose a service call is being invoked to store data in a cloud database. The application might not need an acknowledgement right away that the storage operation has completed. Therefore, the service call can be made asynchronously, allowing the application to continue executing while the remote storage operation is still executing. A callback can be registered in the ListenableFuture which implements the asynchronous call to the service; this callback can provide a notification that the service call has completed.

We can further mitigate the overhead for accessing a remote service by having a local service provide similar functionality. For example, an application may invoke a service to store data in a remote cloud database. Our rich SDK also provides the ability to store data locally in a file system, database, or cache. Local storage will generally incur significantly lower latency and will often be sufficient. Data can be stored locally for performance reasons and occasionally stored in the cloud database to avoid seriously slowing down performance. If the client will often be disconnected from the network or experience poor connectivity (which can be the case for a mobile client), the local storage service can be used during periods of poor connectivity. When connectivity is restored, local storage can be properly synchronized with remote storage.

The rich SDK might also have information on the estimated monetary cost to invoke a service. This can be factored into making decisions on which services to invoke. The rich SDK might also have information on the quality of responses produced by a service. Users can provide methods to the rich SDK which evaluate the quality of data provided by a service. The rich SDK can pick a service to invoke by considering the latency, monetary cost, and quality of data that it has for each service. It can weight each of these three factors using a set of default weights or weights provided by the user.

In order to rank services, we assign a score to the service which considers the latency, monetary cost, and quality of data produced by the service. One possible formula for the score is the following:

$$S = \alpha_1 * r + \beta_1 * c - \gamma_1 * q \quad (1)$$

In this formula, S is the score used to determine the rank of a service, r is its predicted response time, c is its predicted monetary cost, and q is a number representing the predicted quality of data returned by the service. Higher values of q indicate higher quality. The predicted values are based on past data that the rich SDK has collected from previous executions of the service. If there is insufficient past data to predict any of these values, then default values are used which can be the average value for similar services, the median value for similar services, or default values provided by the user. The quantities α_1, β_1, and γ_1 are weights which represent the relative importance of response time, monetary cost, and quality of data. These weights can be specified by the user.

An alternative score for a service normalizes predicted response times, monetary costs, and quality of data returned by the service to values between 0 and 1. Suppose that r, c, and q are all nonnegative. Then we can use the following formula for the score:

$$S_n = \alpha_2 * r/r_{max} + \beta_2 * c/c_{max} - \gamma_2 * q/q_{max} \quad (2)$$

where r_{max}, c_{max}, and q_{max} are the highest values for r, c, and q respectively among all services having similar functionality. The rich SDK allows scores to be assigned to services using Equation 1, Equation 2, or a customized formula provided by the user. The rich SDK can rank services having similar functionality by sorting the services in increasing order by score. The service with the lowest score is the most desirable one. Thus, lower scores correspond to higher service ranks.

The rich SDK is written in Java and includes Java methods which encapsulate the code making calls to the Web and cloud services. Our rich SDK makes the appropriate HTTP calls and converts the responses into Java data types which can easily be accessed by user programs. In order to allow programs written in other languages to access the rich SDK, the rich SDK can expose an HTTP interface allowing applications written in other languages to use it.

We have also developed enhanced data store clients which provide key features for applications accessing cloud data stores such as client-side caching, encryption, and compression [11]. Our work on enhanced data store clients is complementary but distinct from the work described in the current paper.

2.1. Handling Failures and Multiple Services with Similar Functionality

Remote services can sometimes be unresponsive. If a service is unresponsive, the rich SDK has the ability to retry a service multiple times. The number of retries can be specified by the user. In some cases, multiple services handling the same functionality may be available. The service ranking capabilities of the rich SDK can be used to determine the order for trying different services in order to find a responsive one. It would generally be preferable to

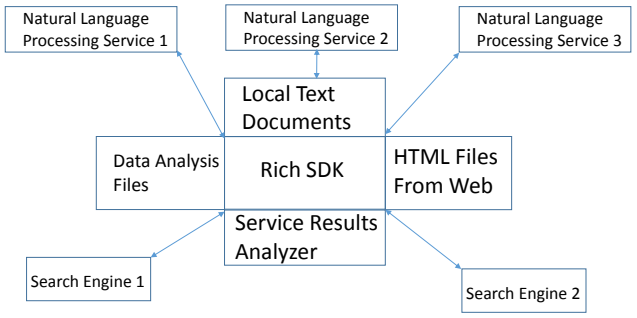

Figure 3: This figure depicts several key features provided by our rich SDK for supporting and enhancing natural language understanding services.

start with higher ranked services and continue with lower ranked services until a responsive service is found. The number of times to retry each service before moving on to the next one can be specified by the user and may be different for different services.

If we have multiple services providing similar functionality, it is sometimes desirable to invoke more than one service instead of just picking a single one. For example, it may be desirable to store the same data on different cloud databases. This provides redundancy. If one database is unavailable for some reason, one or more other databases should be available. In some cases, it may be important to invoke multiple services with similar functionality in order to analyze and possibly combine output from the services. For example, different services exist for natural language understanding. It may be desirable to aggregate or compare the results from different language understanding services. More specifically, if a text document is being analyzed for named entity recognition or relationship extraction, it may be desirable to use multiple named entity recognition or relationship extraction services. The results from these services could be combined. If the results are inconsistent, the application could assign a higher degree of confidence to entities or relationships which are identified by more services. The application might also be comparing the output of these services to determine how good they are.

The service ranking capabilities provided by the rich SDK can be used to determine which services should be invoked in the above scenario. It would generally be preferable to invoke the higher ranked services. Our rich SDK allows the services to be invoked in multiple ways. The services can all be invoked synchronously without using a separate thread for the service calls. This avoids thread synchronization issues but can slow down the application since the application is blocked during each service invocation, and services might have high latencies. Another option is to use a separate thread to execute all of the service calls synchronously. This results in parallel execution between the service calls and the application and thus avoids blocking the application. However, synchronization may be needed for the application to know when the service calls have completed. This can be provided via the use of the ListenableFuture interface described earlier. This approach can still have high latency due to the fact that there is no parallelism in service calls. In order to get around this bottleneck, multiple threads can be used to make parallel service calls. It is not expected that the number of threads would be too high since the number of services with similar functionality is not likely to be high. Nevertheless, to prevent the number of threads from becoming too large in corner cases, we use thread pools of limited size.

2.2. Natural Language Understanding Support

A key feature of cognitive applications is the ability to understand language. Having this capability is critically important for data analytics applications. Many data sources are in the form of unstructured text data, so having the capability to process such data in intelligent ways is critically important. Natural language understanding services, as well as speech recognition, are available from several companies including IBM, Amazon, Google, and Microsoft. The availability of such services makes it feasible to develop applications which use the services to provide advanced features. Our rich SDK provides a number of features to enhance natural language recognition services (Figure 3).

Natural language understanding services typically expose an API wherein they are passed a single text document and return the results from analyzing the single document. Our rich SDK provides support for analyzing multiple documents and aggregating the results from the analysis. We provide the ability to perform Web searches, analyze all of the documents returned by a Web search, and aggregate the results from all analyzed documents. Users can use a variety of search engines such as Google, Bing, and Yahoo! Searches can also be restricted to news stories on Google or Bing. If the natural language understanding service has the ability to analyze Web documents specified by a URL, the rich SDK can pass the URLs returned by the Web search to the service. The rich SDK generally has to send each URL in a separate request because the APIs generally only support analysis of a single document at a time.

Our rich SDK can also fetch HTML documents corresponding to URLs returned from a Web search. These HTML documents can then be passed to natural language understanding services. The rich SDK provides convenient methods for storing HTML documents fetched from Web searches. Storing the documents locally is useful if they need to be analyzed or examined multiple times. Performance is considerably improved since the documents do not have to be fetched again from a remote source. Furthermore, Web documents can become unavailable at a later time, and the results from a Web search can change over time. It is thus valuable to be able to store all of the documents from a particular Web search along with the query itself and the time the query was made.

Our rich SDK has features for passing multiple files to a service and aggregating the results. For example, if a directory contains all HTML documents identified by

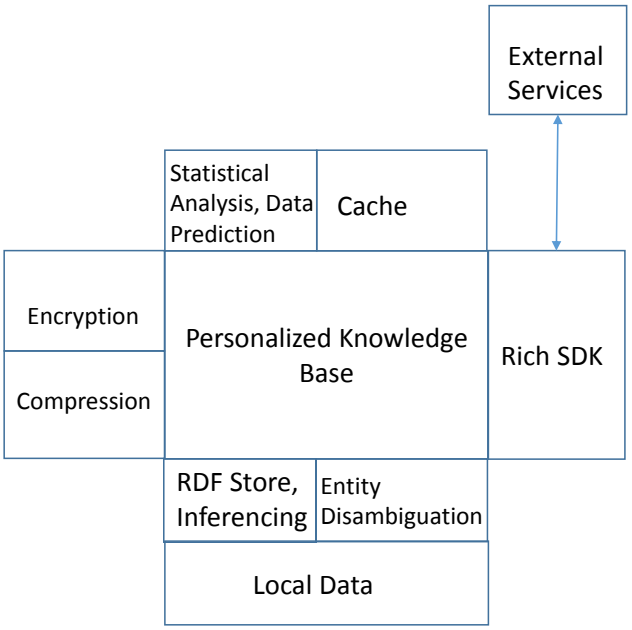

Figure 4: This figure depicts several key features provided by our personalized knowledge base.

responses to a search engine query made at a certain point in time, the rich SDK can pass each file to a natural language understanding system and aggregate the results. The rich SDK can do the same for other directories containing related text files.

Similar types of analyses can be performed on other types of data such as image files. Search engines can identify images matching a query; these images can be passed to an image analysis service and/or stored locally.

Language understanding services can extract things such as named entities, keywords, concepts, taxonomies, and sentiment from a document. Our rich SDK aggregates data from several documents for each of these things. For example, we can compute the frequency with which a certain named entity or keyword appears in a set of documents returned by a query to a Web search engine. Named entities are disambiguated [12], while keywords are not. Our results can thus indicate which named entities and keywords are most relevant to the search query.

Sentiment analysis [13] can provide a quantitative value for a document indicating how positive or negative the document is. However, an entire document may describe several different entities. It is often more meaningful to obtain sentiment scores for individual entities rather than an entire document. IBM offers services through its Watson Developer Cloud which provide sentiment analysis for individual entities and keywords, as well as for documents as a whole. Using the rich SDK, it is possible to determine aggregate sentiment scores for all named entities and keywords across several documents. We have been using the rich SDK to determine how favorably people, companies, and other entities are represented on the Web.

Our rich SDK has the ability to persistently store and later retrieve data returned from invoking a language understanding service on a document. That way, each document only has to be analyzed once. This avoids the latency for invoking a service on the same document multiple times. Invoking services unnecessarily can also have a monetary cost. For some services, the client may have a limited quota of service invocations in a time period (e.g. one day). There is thus an incentive to limit the number of service invocations.

2.3. Access to Data Services

There are a wide variety of external data sources accessible from the Web. These include knowledge bases such as DBpedia, Wikidata, and Yago. DBpedia [1], [2] is a crowd-sourced community effort for extracting structured information from Wikipedia. Online versions of DBpedia are available which can be queried over HTTP. Alternatively, the entire data set can be downloaded. Wikidata is a knowledge base maintained by the Wikimedia Foundation [3], [4]. YAGO (Yet Another Great Ontology) [5] is a knowledge base developed at the Max Planck Institute for Informatics. The information in Yago is derived from Wikipedia, WordNet [14], [15], and GeoNames. There are also a wide variety of other services providing stock, economic, geographical, climate, and many other types of data.

Such data services can be used with our rich SDK to provide valuable data for user applications. The rich SDK allows data obtained from these data services to be cached and stored locally. This can improve performance by avoiding downloading the same data multiple times.

3. A Personalized Knowledge Base

We have developed a personalized knowledge base on top of our rich SDK which allows users to store and analyze personal data and provides the ability to access and analyze publically available data through our SDK. Personal data is data which is not publically available. It may have confidentiality requirements, although this does not have to be the case. The personalized knowledge base is implemented in Java.

Data maintained in the personal knowledge base can exist in a variety of forms and may be structured or unstructured. The data may include text, videos, and images which can be analyzed using cognitive cloud services via our rich SDK. The personal knowledge base can store data persistently in a variety of forms including files, relational database management systems (RDBMS), key-value stores, and RDF [9] triple stores (Figure 4). Support is provided for reading and writing comma-separated value (CSV) files. MySQL [16] is used for the RDBMS, and Apache Jena [17], [18] is used for the RDF store. Other relational database management systems and triple stores could be used in place of MySQL and Apache Jena.

Data can also be stored within cloud or other types of remote data stores. The personalized knowledge base

uses enhanced data store clients which reduce the latency for accessing remote data stores via caching [11], [19]. Enhanced data store clients also allow data to be encrypted and compressed before being sent to a remote data store.

The personalized knowledge base provides methods to allow data to be converted to different formats. Data in CSV files can be added to a relational database table in MySQL or an RDF model in Jena. RDF models consist of statements. A statement has three parts: a *subject, predicate,* and *object.* For example, in the statement *"The Java HashMap class implements the Java Map interface"*, *"Java HashMap class"* is the subject, *"implements"* is the predicate, and *"Java Map interface"* is the object. A Jena statement can be added to a MySQL table. Conversely, MySQL tables can be converted to Jena statements.

One of the advantages to storing data as Jena statements is that Jena can perform reasoning on the data that it is storing. Jena provides a number of predefined reasoners including the following:

- A transitive reasoner with support for storing and traversing class and property lattices.
- An RDF Schema [20] rule reasoner which implements a configurable subset of the RDF Schema entailments [21].
- Reasoners which support an incomplete implementation of the OWL/Lite subset of the OWL/Full language [22].
- A generic rule reasoner that supports user-defined rules. This reasoner supports forward chaining, tabled backward chaining, and hybrid execution strategies.

Jena includes a SPARQL [23], [24] query engine which the personalized knowledge base uses to query data sources such as DBpedia.

The personalized knowledge base incorporates data from multiple sources. Named entity disambiguation [12] is a key problem, since the same entity can be referred to in different ways. For example, the country *United States of America* is also referred to as *USA, US, United States, America,* and even *the states.* If we use a simple string matching algorithm to identify entities, then we might mistakenly conclude that the string *"United States of America"* refers to a different country than the string *"USA"*. The personalized knowledge base can use natural language understanding services provided by IBM's Watson Developer Cloud to disambiguate entities such as country names. For example, the sentence *"The US is a country"* can be passed to IBM's Watson Developer Cloud. IBM's Watson Developer Cloud correctly disambiguates the country referenced by the string *US* and returns several URLs associated with the US including the URL for the main Web site of the US government and URLs for DBpedia and YAGO Web pages containing information about the US:

```
"website": "http://www.usa.gov/",
"dbpedia": "http://dbpedia.org/resource/
United_States",
"yago": "http://yago-knowledge.org/
resource/United_States"
```

Using this approach, our personalized knowledge base can usually correctly identify a unique country referenced by a string. A unique country ID is the preferred way to identify each country within the personalized knowledge base. This prevents the proliferation of redundant database entries which could occur if strings corresponding to country names are not disambiguated.

Other services can be used for disambiguating entities as well. The problem can get complex depending upon the type of entities. For example, there are a large number of human diseases. Furthermore, there are several different methods for naming diseases. Data sets related to diseases might use different conventions for naming diseases. Depending on the naming conventions used, there might not be a service or tool available which can accurately disambiguate the disease names.

For domains for which there are no existing services or tools to help with entity disambiguation, users can provide their own files which identify synonyms which map to the same entity.

The personalized knowledge base also includes a spell checker. While there are many spell checking services which are offered over the Web, the spell checker included with the knowledge base is generally faster as it avoids the overheads of remote communication. Some online spell checkers also cost money.

The personalized knowledge base also has the ability to perform statistical and mathematical analysis on numerical data. Regression analysis can be used to predict new data values from existing values. We use the Apache Commons Math library [25] for mathematical and statistical analysis. The personalized knowledge base can also output data in files (e.g. CSV files) which can be analyzed by other data analysis tools such as MATLAB, Excel, Python programs, R, programs, etc. These external tools can output the results in CSV files which the personalized knowledge base can ingest.

One powerful way of using mathematical analysis is to store the key mathematical results as RDF statements. The RDF store has the ability to perform inferencing on the statements in the RDF store to generate new statements. Therefore, mathematical analysis combined with inferencing on the RDF store can generate new knowledge beyond that produced by just the mathematical analysis itself (Figure 5).

As the RDF store infers new facts, these facts can be converted to other formats. For example, the facts can be stored in a relational database or a text file. The ability to convert data between different formats is a key property of our personalized knowledge base.

The personalized knowledge base provides encryption to preserve data confidentiality. Data can be encrypted before it is stored persistently to prevent the data from being leaked. This is particularly important if the data is being stored in a cloud or other type of remote data store. Even if the remote data store provides encryption, the personal knowledge base

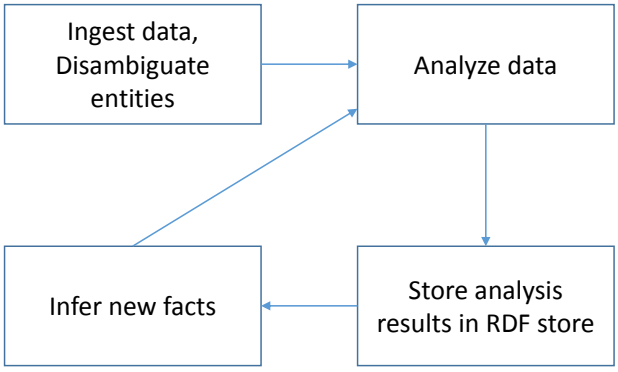

Figure 5: The personalized knowledge base can analyze data in a wide variety of ways including statistical techniques and prediction using regression. Inferencing can be performed on the analysis results to generate more facts.

may need to encrypt the data before sending it over the network to the remote data store if the communication channel with the remote data store is not secure. In addition, if the remote data store is not trusted, then the personal knowledge base might need to encrypt confidential data before sending it to the remote data store even if the remote data store has encryption capabilities and the personal knowledge base is communicating with the remote data store over a secure channel.

Compression can be applied to reduce disk and memory requirements for storing data. Compression by the personal knowledge base can be important for storing data in a cloud or other type of remote data store even if the remote data store provides compression. If the personal knowledge base compresses data before sending it to the remote data store, less network bandwidth will be required. Some cloud data stores charge users based on space consumption. Therefore, compression by the personal knowledge base can save money, even if the cloud data store provides compression.

The personalized knowledge base tries to accommodate scenarios where the computer (s) on which it runs may be disconnected from the network. Caching and local storage can be used when remote data sources and services are not accessible. The personalized knowledge base has data analytics capabilities which it can execute locally without relying on external services. When the personalized knowledge base becomes disconnected from a cloud data store for a considerable period of time, it may be appropriate to synchronize the contents of local storage and the cloud data store after connectivity between the personalized knowledge base and the cloud data store is re-established.

4. Related Work

There are several cognitive services accessible over the Web, including those from IBM [26], Microsoft [27], Amazon [28], and Google [29]. There are also several SDKs in support of cognitive and other cloud services such as those from Amazon [6], IBM [7], and Microsoft [8]. Our rich SDK provides several features not provided by previous SDKs.

There have been several papers which compare and rank cloud services. A framework to compare different cloud providers based on user requirements is proposed in [30], [31]. Users can then compare different cloud offerings and select the most appropriate services based on their needs. A personalized ranking prediction framework to predict the QoS ranking of a set of cloud services is proposed in [32], [33]. The approach takes advantage of the past usage experiences of other users for making personalized ranking predictions for the current user. A brokerage-based architecture where cloud brokers are responsible for service selection is presented in [34]. It assumes that a cloud broker exists which has a contract with the cloud service providers. The cloud broker collects properties of cloud service providers (such as service type, unit cost, and available resources) and user service requirements to rank services for a user request. Cloud service selection is performed in [35] by aggregating subjective assessments from cloud consumers and objective performance assessments from a trusted third party. The authors apply a fuzzy simple additive weighting system to normalize and aggregate all different types of subjective and objective attributes of a cloud service. The authors also propose a method for filtering unreasonable feedback from biased or malicious users. Unlike our work, none of these papers has a focus on cognitive services. A survey of cloud service selection techniques with many references to past work in the area is presented in [36].

A number of knowledge bases containing a broad set of facts are publically available. DBpedia [1], [2] is a crowd-sourced community effort for extracting structured information from Wikipedia. Online versions of DBpedia are available which can be queried over HTTP. Alternatively, the entire data set can be downloaded. Wikidata is a knowledge base maintained by the Wikimedia Foundation [3], [4]. YAGO (Yet Another Great Ontology) [5] is a knowledge base developed at the Max Planck Institute for Informatics. The information in Yago is derived from Wikipedia, WordNet [14], [15], and GeoNames. OpenCyc [37] is a knowledge base based on the Cyc [38] artificial intelligence project which assembles a comprehensive ontology and knowledge base of everyday common sense knowledge. The project has a long history going back to 1984. Freebase [39] was a knowledge base of structured data which is now inactive. Much of its content has been transferred to Wikidata.

Named entity disambiguation is an important feature for the personalized knowledge base. There is a considerable amount of past work in this area [12], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50].

Enhanced clients applicable to cloud and other remote data stores are presented in [11]. These enhanced clients are used by our personalized knowledge base.

5. Conclusion

This paper has looked at the problem of providing support for applications which access cognitive services over

the Web. We have presented a software development kit (SDK) which provides several capabilities which previous SDKs do not offer. Our rich SDK can collect data on services related to performance, availability, and the quality and accuracy of responses. It can rank services providing similar functionality and help an application select the most appropriate service (s) to invoke. The rich SDK can also handle service request failures and retry failed requests for services. A key use case for our rich SDK is to help applications use intelligent services which understand language and perform visual recognition. Users can analyze several text documents and aggregate results from this analysis. The rich SDK combines Web searches with natural language processing and has features for analyzing results from natural language understanding services and storing these results persistently.

This paper also presents a personal knowledge base built on top of our rich SDK. The personal knowledge base allows data to be stored in multiple ways. We provide relational database management systems (RDBMS), key-value stores, RDF triple stores, and support for storing data organized as comma-separated values (CSV) in files. The personal knowledge base can also convert data from one format to another. Our system can analyze data for patterns and perform predictive analytics; it also provides inferencing capabilities.

There are a number of ways to extend this work. For the rich SDK, more sophisticated methods can be used for evaluating the quality of responses provided by services. For the personalized knowledge base, one of the key issues is how to deal with data sources which contain data which may not be completely accurate or may not be consistent with data obtained from other sources. We would like ways of determining accuracy levels of data stored within the personalized knowledge base, using these accuracy levels during the process of inferring new facts, and assigning accuracy levels to newly inferred facts.

References

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A Nucleus for a Web of Open Data," in *Proceedings of the 6th International Semantic Web Conference (ISWC 2007) and 2nd Asian Semantic Web Conference (ASwC 2007)*, November 2007, pp. 722–735.

[2] DBpedia, "About DBpedia," http://wiki.dbpedia.org/about.

[3] D. Vrandecic and M. Krotsch, "Wikidata: A Free Collaborative Knowledge Base," *Communications of the ACM*, vol. 57, no. 10, pp. 78–85, October 2014.

[4] Wikimedia Foundation, "Wikidata:Introduction," https://www.wikidata.org/wiki/Wikidata:Introduction.

[5] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A Core of Semantic Knowledge Unifying WordNet and Wikipedia," in *Proceedings of the 16th International World Wide Web Conference (WWW 2007)*, May 2007, pp. 697–706.

[6] Amazon, "AWS SDK for Java," https://aws.amazon.com/sdk-for-java/.

[7] IBM, "Watson Developer Cloud Java SDK," https://github.com/watson-developer-cloud/java-sdk.

[8] Microsoft, "Microsoft Azure Downloads," https://azure.microsoft.com/en-us/downloads/.

[9] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," https://www.w3.org/TR/rdf11-concepts/.

[10] Google, "ListenableFutureExplained," https://github.com/google/guava/wiki/ListenableFutureExplained.

[11] A. Iyengar, "Providing Enhanced Functionality for Data Store Clients," in *Proceedings of the IEEE 33rd International Conference on Data Engineering (ICDE 2017)*, April 2017.

[12] S. Cucerzan, "Large-Scale Named Entity Disambiguation Based on Wikipedia Data," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, June 2007, pp. 708–716.

[13] B. Pang, L. Lee *et al.*, "Opinion mining and sentiment analysis," *Foundations and Trends® in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.

[14] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, November 1995.

[15] Princeton, "WordNet: A lexical database for English," https://wordnet.princeton.edu/.

[16] MySQL, "MySQL home page," https://www.mysql.com/.

[17] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the Semantic Web Recommendations," in *Alternate Track Papers and Posters of the 13th International World Wide Web Conference (WWW 2004)*, May 2004, pp. 74–83.

[18] Apache, "Apache Jena home page," https://jena.apache.org/.

[19] IBM, "Data Store Client Library," https://developer.ibm.com/open/data-store-client-library/.

[20] D. Brickley and R. Guha, "RDF Schema 1.1," https://www.w3.org/TR/rdf-schema/.

[21] P. J. Hayes and P. F. Patel-Schneider, "RDF 1.1 Semantics," https://www.w3.org/TR/rdf11-mt/.

[22] M. K. Smith, C. Welty, and D. L. McGuinness, "OWL Web Ontology Language Guide," https://www.w3.org/TR/owl-guide/.

[23] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language," https://www.w3.org/TR/sparql11-query/.

[24] J. Perez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPARQL," *ACM Transactions on Database Systems*, vol. 34, no. 3, August 2009.

[25] Apache, "Commons Math: The Apache Commons Mathematics Library," http://commons.apache.org/proper/commons-math/.

[26] IBM, "Watson Developer Cloud," https://www.ibm.com/watson/developercloud/.

[27] Microsoft, "Microsoft Cognitive Services," https://www.microsoft.com/cognitive-services.

[28] Amazon, "Amazon AI," https://aws.amazon.com/amazon-ai/.

[29] Google, "Cloud Natural Language API," https://cloud.google.com/natural-language/.

[30] S. K. Garg, S. Versteeg, and R. Buyya, "Smicloud: A framework for comparing and ranking cloud services," in *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*. IEEE, 2011, pp. 210–218.

[31] ——, "A framework for ranking of cloud computing services," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1012–1023, 2013.

[32] Z. Zheng, Y. Zhang *et al.*, "Cloudrank: A qos-driven component ranking framework for cloud computing," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 184–193.

[33] Z. Zheng, X. Wu, Y. Zhang, M. R. Lyu, and J. Wang, "Qos ranking prediction for cloud services," *IEEE transactions on parallel and distributed systems*, vol. 24, no. 6, pp. 1213–1222, 2013.

[34] S. Sundareswaran, A. Squicciarini, and D. Lin, "A brokerage-based approach for cloud service selection," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 558–565.

[35] L. Qu, Y. Wang, and M. A. Orgun, "Cloud service selection based on the aggregation of user feedback and quantitative performance assessment," in *Services Computing (SCC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 152–159.

[36] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, and E. Chang, "Cloud service selection: State-of-the-art and future research directions," *Journal of Network and Computer Applications*, vol. 45, pp. 134–150, 2014.

[37] M. A. Sicilia, E. García, S. Sánchez, and E. Rodríguez, "On integrating learning object metadata inside the opencyc knowledge base," in *Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 900–901.

[38] D. B. Lenat, "Cyc: A large-scale investment in knowledge infrastructure," *Communications of the ACM*, vol. 38, no. 11, pp. 33–38, 1995.

[39] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. AcM, 2008, pp. 1247–1250.

[40] R. C. Bunescu and M. Pasca, "Using encyclopedic knowledge for named entity disambiguation." in *Eacl*, vol. 6, 2006, pp. 9–16.

[41] P. N. Mendes, M. Jakob, A. García-Silva, and C. Bizer, "Dbpedia spotlight: shedding light on the web of documents," in *Proceedings of the 7th international conference on semantic systems*. ACM, 2011, pp. 1–8.

[42] J. Hoffart, M. A. Yosef, I. Bordino, H. Fürstenau, M. Pinkal, M. Spaniol, B. Taneva, S. Thater, and G. Weikum, "Robust disambiguation of named entities in text," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2011, pp. 782–792.

[43] D. Milne and I. H. Witten, "Learning to link with wikipedia," in *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 2008, pp. 509–518.

[44] L. Ratinov, D. Roth, D. Downey, and M. Anderson, "Local and global algorithms for disambiguation to wikipedia," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*. Association for Computational Linguistics, 2011, pp. 1375–1384.

[45] M. Dredze, P. McNamee, D. Rao, A. Gerber, and T. Finin, "Entity disambiguation for knowledge base population," in *Proceedings of the 23rd International Conference on Computational Linguistics*. Association for Computational Linguistics, 2010, pp. 277–285.

[46] X. Han and J. Zhao, "Named entity disambiguation by leveraging wikipedia semantic knowledge," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 215–224.

[47] J. Hassell, B. Aleman-Meza, and I. B. Arpinar, "Ontology-driven automatic entity disambiguation in unstructured text," in *International Semantic Web Conference*. Springer, 2006, pp. 44–57.

[48] A. Moro, A. Raganato, and R. Navigli, "Entity linking meets word sense disambiguation: a unified approach," *Transactions of the Association for Computational Linguistics*, vol. 2, pp. 231–244, 2014.

[49] R. Usbeck, A.-C. N. Ngomo, M. Röder, D. Gerber, S. A. Coelho, S. Auer, and A. Both, "Agdistis-graph-based disambiguation of named entities using linked data," in *International Semantic Web Conference*. Springer, 2014, pp. 457–471.

[50] S. Zwicklbauer, C. Seifert, and M. Granitzer, "Robust and collective entity disambiguation through semantic embeddings," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 2016, pp. 425–434.