Journal of Cheminformatics

RESEARCH ARTICLE

Open Access

SIA: a scalable interoperable annotation server for biomedical named entities

Johannes Kirschnick[†], Philippe Thomas[†], Roland Roller and Leonhard Hennig[*]

Abstract

Recent years showed a strong increase in biomedical sciences and an inherent increase in publication volume. Extraction of specific information from these sources requires highly sophisticated text mining and information extraction tools. However, the integration of freely available tools into customized workflows is often cumbersome and difficult. We describe SIA (Scalable Interoperable Annotation Server), our contribution to the BeCalm-Technical interoperability and performance of annotation servers (BeCalm-TIPS) task, a scalable, extensible, and robust annotation service. The system currently covers six named entity types (i.e., chemicals, diseases, genes, miRNA, mutations, and organisms) and is freely available under Apache 2.0 license at https://github.com/Erechtheus/sia.

Keywords: Text mining, Annotation service, Robustness, Scalability, Extensibility

Introduction

A vast amount of information on biomedical processes is scattered over millions of scientific publications. Manual curation of this information is expensive and cannot keep up with the ever increasing volume of biomedical literature [1]. To this end, several sophisticated natural language processing tools have been proposed to assist professionals in finding specific information from texts. Many of these highly specialized tools are provided as open source projects to the community. However, the integration of state-of-the-art open source extractors into customized text-mining workflows is often difficult and cumbersome [2, 3]. Standardized interchange formats, such as BioC [4], enable the exchange of text mining results but the initial set-up of these tools is still an unsolved issue. Exposing tools via public web services implementing common specifications bypasses this problem and allows a code-agnostic integration of specific tools by providing an interoperable interface to third parties. This enables simple integration, comparison, and aggregation of different state-of-the-art tools. In this publication we present SIA, our contribution to the BeCalm-Technical interoperability and performance of annotation servers (BeCalm-TIPS) task [5]. SIA is a robust, scalable, extensible, and generic framework to combine multiple named entity recognition tools into a single system.

The publication is organized as follows: First, we briefly introduce the BeCalm-TIPS task and its requirements. We then give an overview of the SIA system architecture, followed by a detailed description of the implementation and the error handling features. This is followed by a scalability experiment conducted on a large dump of PubMed articles and a discussion of the results. We end with a summary and a future work section.

BeCalm-TIPS task overview

The following section provides a short introduction to the BeCalm-TIPS task, focusing on the payloads annotation servers had to accept and respond with. A detailed description of the task is available in [5].

The task set out to define a testbed for comparing different annotation tools by making them accessible via public web endpoints which exchange standardized JSON messages. It required participants to register their endpoint and a set of supported named entity types with a system managed by the task organizers. Over the course of the task, this endpoint received a number of annotation requests. Each request was not required to be processed interactively, just the message reception had to be

*Correspondence: leonhard.hennig@dfki.de
†J. Kirschnick and P. Thomas contributed equally to this work
DFKI Language Technology Lab, Alt-Moabit 91c, Berlin, Germany

acknowledged. Once the annotations were generated by the annotation server, they had to be sent back to a dedicated endpoint—via a separate HTTP request.

Listing 1: JSON payload excerpt for an annotation request

```
{"documents":
   [{"document_id": "BC1403854C", "source":"PUBMED"}, ...],
 "types": ["DISEASE", "MUTATION", "MIRNA"],
 "communication_id": 1581}
```

Listing 1 shows an excerpt of the JSON payload for an annotation request. It consists of a list of document identifiers and their respective source. As no text was transmitted, participants where required to implement their own text retrieval component to fetch the title, abstract and potentially full text for each document prior to processing. A type field specified the list of named entities to be identified. A unique communication identifier was passed along, which had to be included in any outgoing messages in order to correlate individual requests and responses.

Listing 2: JSON payload excerpt for an annotation response

```
[{"document_id":"BC1403855C", "section":"A",
  "init":410,   "end":419,    "score":1.0,
  "type":"DISEASE", "annotated_text":"periosteum" }, ...]
```

Once the annotation server acknowledged the reception of a request it had a specified amount of time to respond. Listing 2 shows a snippet of such a response. It contains a list of detected annotations across all requested documents, identifying the text source section (abstract *A* or title *T*), the start and end positions within it, a confidence score, and the extracted named entity type as well as the annotated text itself.

The task merely specified the required input—as well as output schemata and gave participants full control over the implementation of their system as well as which annotation types they wanted to support.

SIA: general architecture

This section describes the architecture of SIA, our contribution to the BeCalm-TIPS task. Figure 1 shows a high level overview of the general architecture, which was designed around the following three main goals:

1. **Scalability** The ability to handle large amounts of concurrent requests, tolerating bursts of high request rates over short periods of time.
2. **Robustness** Temporary failures (e.g., networking problems or server failures) should be handled transparently and not lead to dropped requests.
3. **Extensibility** Enable simple integration of arbitrary NLP tools to reduce initial burden for providing an annotation service.

To achieve these goals, SIA is split into three components, the **front end**, **back end**, and **result handling**, respectively. The front end handles the interactive aspects of the system, while the other components implement the system's non-interactive elements.

To connect these components, we opted for a message based architecture, which links each component to a central message bus, over which they exchange messages. Incoming annotation requests are translated into messages by the front end, and subsequently processed by the back end. Once processing is finished the final result is handled by the result handler. To this end, SIA defines a configurable message flow for each message, which incorporates fetching raw texts, running a set of annotators, aggregating the results and forwarding them to a result handler. The configuration defines the actual processing steps, the set of annotator components to use, which document fetchers to enable and how to deal with the results. For example, a processing flow could fetch PubMed articles from a public endpoint, handle all requests for *Mutations* with the SETH [6] tagger and send annotation results back to the requester. The overall processing flow is expressed as an ordered sequence of message transformation and aggregation steps, while the configuration allows to extend the actual processing flow with new annotator and document fetcher components. Interested readers are referred to Enterprise Integration Patterns [7] for a detailed discussion of the different

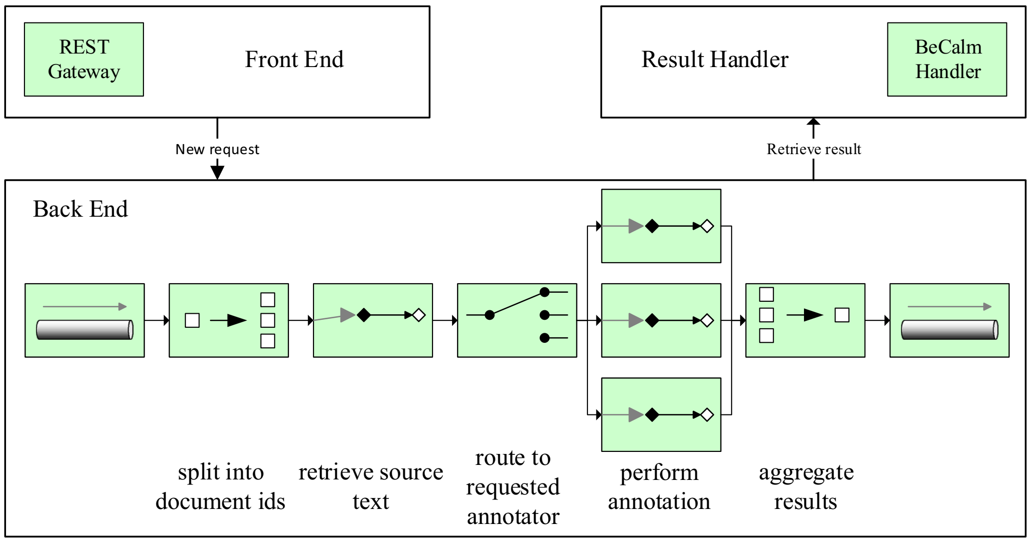


Fig. 1 General architecture of SIA. The front end handles new requests and forwards them to the back end over a message bus. Each message is transformed through a series of components, which in turn are connected via named queues. The result handler collects the annotation responses and returns them to the calling client

message handling and transformation strategies that SIA employs.

To handle messages, persistent queues are defined as input and output buffers for all components, where a subsequent component consumes from the previous component's output queue. These queues are stored for the entirety of the system's lifetime. This architecture further provides fault tolerant and scalable processing. Fault tolerance is enabled through component wise acknowledgment of each successful message processing, which allows replaying all unacknowledged messages during system recovery, while scalability is achieved through component replication and round robin based message forwarding for increased message throughput.

Messages, the data objects in our architecture, carry information through the system and are composed of a Header and Payload part. The Header contains meta information, such as expiry date, global ids and requested annotation types, and is used by the system to route messages to the respective consumers. The Payload contains the actual data to be processed.

Implementation details

SIA is implemented in Java and uses RabbitMQ [8] as its message bus implementation. In the following each individual component of SIA is described in detail.

Front end

The front end encapsulates the annotation processing for the clients and serves as the entry point to the system. Currently it provides a REST endpoint according to the Becalm-TIPS task specification. Other entry points, such as interactive parsing can easily be added. Incoming requests are translated into messages and forwarded to an input queue. This way, the overall processing in the front end is very lightweight and new requests can be handled irrespectively of any ongoing annotation processing. Furthermore, the back end does not need to be online at the time of a request, but instead could be started dynamically based on observed load.

To handle multiple concurrent requests with varying deadlines, we make use of the fact that the input queue is a priority queue, and prioritize messages with an earlier expiry date. Already running requests will not be canceled, the priority is just used as a fast path to the front of the queue. The message expiry date, as provided by the calling clients, is translated into a message priority using the currently processed messages and their deadlines as well as past elapsed processing time statistics to estimate the individual message urgency.

The front end also handles validation and authorization, which moves this logic into a central place. Furthermore, the front end provides a monitoring entry point into the system, reporting computation statistics, such as request rates, recent document types as well as back end processing counters, for display in dashboards and for observing the current health of the system.

Back end

The back end is concerned with fetching documents from the supported corpus providers, calling the requested annotators for each resulting text fragment, aggregating the results and feeding them to a result handler. It is modeled using a pipeline of message transformations, which subsequently read from message queues and post back to new ones. The message flow starts by reading new requests from the input queue, which is filled by the front end. The front end does not communicate directly with the back end, but instead the input queue is used as a hand over point. Since a single annotation request, in the case of the Becalm-TIPS task specification, may contain multiple document ids, incoming messages are first split into document-level messages. Splitting takes one message as input and generates as many individual messages as there are document ids specified. The raw text for each document is then retrieved by passing the messages through corpus adapters. The outcome is the retrieved text, separated into fields for abstract, title and potentially full text.

Raw texts messages are then delivered to registered annotators using a scatter-gather approach. Each message is duplicated (scattered) to the respective input queue of a qualified annotator. To find the annotator, the required annotator type per message is translated into a queue name, as each annotator has a dedicated input queue. Upon completion all resulting annotation messages are combined together (gathered) into a single message. This design allows to add new annotators by registering a new input queue and adding it to the annotation type mapping. This mapping is also exposed as a runtime configuration, which allows to dynamically (de-) activate annotators.

The next step in the message flow aggregates all annotation results across all documents that belong to the same request. It is the inverse of the initial split operation, and aggregates all messages sharing the same unique request id into a single one. Overlapping annotations (e.g., from different annotator components) are merged without any specific post processing. This strategy allows end users the highest flexibility as annotations are not silently modified. Finally, the aggregated message is forwarded to the output queue.

While the processing flow is specified in a sequential manner, this does not entail single threaded execution. Each individual transformer, such as a corpus adapter or an annotator, works independently and can be further scaled out, if they present a processing bottleneck. Furthermore, multiple requests can be handled in parallel at different stages of the pipeline. Transacting the message delivery to each transformer and retrying on failure, provides the fault tolerance of the system. Overall, the back end specifies a pipeline of an ordered execution flow and provides two injection points where users, through configuration, can add new functionality with additional corpus adapters or new annotation handlers.

To increase the throughput of the back end, multiple instances of SIA can be started on different machines, where each instance would process requests in a round robin fashion.

Supported annotators

To illustrate the extensibility of our approach, we integrated named entity recognition (NER) components for six different entity types into SIA: mutation names are extracted using SETH [6]. For micro-RNA mentions we implement a set of regular expressions [9], which follow the recommendations for micro-RNA nomenclature [10]. Disease names are recognized using a dictionary lookup [11], generated from UMLS disease terms [12], and by using the DNorm tagger [13]. Chemical name mentions are detected with ChemSpot [14], Organisms using Linnaues [15] and Gene mentions using Banner [16].

Listing 3 shows the general interface contract SIA is expecting for each annotator. Each annotator receives an input text and is simply expected to return a set of found annotations. Thus integrating any of the aforementioned annotators, as well as new ones, is as simple as implementing this interface and registering a new queue mapping.

Annotation handlers can be hosted inside of SIA, within the same process, or externally, in a separate process. External hosting allows to integrate annotation tools across programming languages, operating systems and servers. This is especially useful since most annotators have conflicting dependencies that are either very hard or impossible to resolve. For example, ChemSpot and DNorm use different versions of the Banner tagger which make them candidates for external hosting. Multiple servers can also be used to increase the available resources for SIA, e.g., when hosting all annotators on the same machine exceeds the amount of available memory.

Corpus adapters

SIA contains corpus adapters for PubMed, PMC, and the BeCalm patent- and abstract servers, which communicate to external network services. These components are represented as transformers, which process document ids and return retrieved source texts. They are implemented following the interface definition shown in Listing 4 . If an adapter supports bulk fetching of multiple documents, we feed a configurable number of ids in one invocation.

As retrieving the full text translates into calling a potentially unreliable remote service over the network, retry on failure is used in case of recoverable errors. This is backed up by the observation that the most commonly observed error was a temporarily unavailable service endpoint. To spread retries, we use exponential backoff on continuous failures with an exponentially increasing time interval, capped at a maximum (initial wait 1*s*, multiplier 2, max wait 60*s*). If a corpus adapter fails to produce a result after retries are exhausted, we mark that document as unavailable and treat it as one without any text. This allows a trade-off between never advancing the processing, as a document could be part of a set of documents to be annotated, and giving up too early in case of transient errors.

Result handler

The result handler processes the aggregated annotation results from the back end, by consuming from a dedicated output queue. We implemented a REST component according to the TIPS task specification, which posts these annotations back to a dedicated endpoint. Additional handlers, such as statistics gatherer or result archiver, can easily be added.

Corpus adapter fails Each adapter retries, using exponential backoff, to fetch a document before it is marked as unavailable. As the BeCalm-TIPS task does not specify how to signal unavailable documents, these are just internally logged. Any subsequent processing treats a missing document as one with no content.

Annotator fails If an annotator fails on a particular message, this can potentially harm the entire back end when annotators are embedded in the system. As annotators are software components not under the control of the processing pipeline, we catch all recoverable errors and return zero found annotations in these cases—logging the errors for later analysis.

Result Handling fails The BeCalm-TIPS task description expects the result of an annotation request to be delivered to a known endpoint. If this fails, the delivery is retried in a similar manner to the corpus adapter failure handling.

Listing 3: Interface definition for SIA annotators

```
public interface Annotator {
    Set<PredictionResult> annotate(InputText payload);
}
```

Listing 4: Interface definition for SIA corpus adapters

```
public interface CorpusAdapter {
    InputText load(String documentID);
}
```

Failure handling

In the following we describe the failure handling strategies across the different components within SIA.

Invalid requests Client calls with wrong or missing information are handled in the front end using request validation. Such invalid requests are communicated back to the caller with detailed error descriptions.

Backpressure To avoid that a large number of simultaneous requests can temporarily overload the processing system, SIA buffers all accepted requests in the input queue - using priorities to represent deadlines.

Front end fails If the front end stops, new requests are simply not accepted, irrespective of any ongoing processing in the back end.

Back end unavailable Messages are still accepted and buffered when there is enough storage space, otherwise the front end denies any new annotation requests.

Back end fails If the back end stops while there are still messages being processed, these are not lost but retried upon restart. This is enabled by acknowledging each message only upon successful processing per component.

Message expired Clients can define a deadline for results. This is mapped to a time-to-live attribute of each message. This results in automatically dropping any expired messages from the message bus.

Performance test

To test the scalability as well as extensibility of SIA we performed an offline evaluation, focusing on throughput. To this end we extended the front end to accept full text documents and added an identity corpus adapter which simply returns the document text from the request message itself. Furthermore, we added a result handler, which writes all results into a local file. By adding these components, we turned SIA into an offline annotation tool, that can be fed from a local collection of text documents without relying on external document providers.

For the test, we used a dump of 207.551 PubMed articles[1] and enabled all internal annotators (SETH, mirNer,

[1] Using files 922 to 928 from [17].

Table 1 Scalability experiment results

Parallelism	Processing time (s)	Throughput (doc/s)	Improvement
1	8.151	25	
2	4.551	46	1.79×
3	3.412	61	2.39×
4	3.032	68	2.69×
5	2.712	77	3.01×

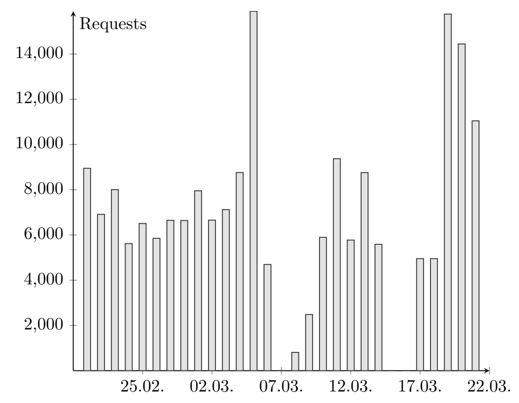
Processing times with varying degree of parallelism, analyzing 207.551 PubMed articles with all internal annotators (SETH, mirNer, Linnaues, Banner, DiseaseNer) and ChemSpot using a single instance of SIA

Linnaues, Banner, DiseaseNer) in a single SIA instance, as well as ChemSpot using the external integration on the same machine. The experiment was run on a Server with 2 Intel Xeon E5-2630 processor (8 threads, 16 cores each) and 256 GB RAM running Ubuntu 12.04. To simulate the scaling behavior, we varied the degree of parallelism used by SIA from 1 to 5 respectively and measured the overall time to annotate all documents. The parallelism controls the number of messages consumed from the input queue simultanously. Table 1 shows the resulting runtimes. When increasing the parallelism we see a decrease of processing times with a speedup of up to 3× compared to single threaded execution. Increasing the parallelism further did not yield lower processing times, as the processing is mainly CPU bound, with a ceiling hit with 5 parallel threads. This highlights that SIA is fully capable of exploiting all available CPU resources, achieving a throughput of more than 70 documents per second. Using the parallelism within SIA furthermore enables to effortlessly provide parallel processing for exiting annotators that are otherwise hard to scale.

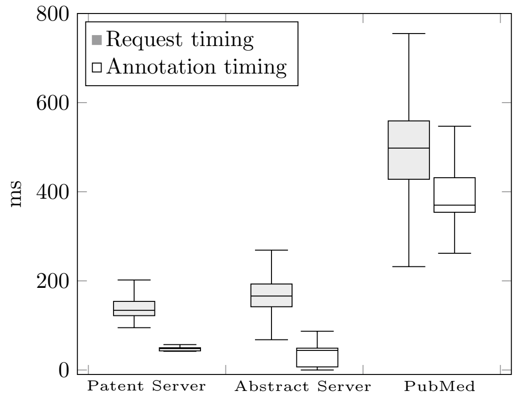
Discussion

SIA itself is very lightweight and runs anywhere given a Java environment and a connection to RabbitMQ. Annotators can be directly embedded or configured to run externally, exchanging messages through the bus. During the BeCalm-TIPS tasks, we deployed SIA into Cloud Foundry, a platform as a service provider, which enables deployments of cloud containers [18]. The front- and back end with embedded result handling were deployed as two separate application containers connected to a hosted instance of RabbitMQ. To limit the resource consumption, we only enabled the SETH, mirNER and DiseaseNER annotators.

Figure 2 shows the received and processed annotation requests over the course of a four week period during the task. It highlights that our system is capable of sustaining a high number of daily requests, with more than 14.000 daily requests received at maximum. Furthermore we observed that the request handling time during these weeks was dominated by individual corpus downloading times, which make up about 50% of the overall processing time. This validates our decision to support bulk downloading of documents, as this amortizes the networking overhead for each document, over a number of documents. Processing each annotation request in total took less than two seconds for the configured annotators. We observed higher annotation times for PubMed articles, which is partially due to higher server response times and the need for more sophisticated result parsing. We also estimated the message bus overhead to about 10%, stemming from individual message serialization and persistence compared to running the annotators stand alone—an acceptable slowdown which is easily compensated by additional parallelism.



a Daily request rates



b Requests processing times for different endpoints

Fig. 2 Processing statistics over a four week period and request times per corpus, reporting complete processing and annotation timings separately

Summary and future work

We described SIA, our contribution to the BeCalm-TIPS task, which provides scalability—through component replication, fault tolerance—through message acknowledgement, and extensibility—through well defined injection points—with a particular emphasis on failure handling. The message-based architecture proved to be a good design blueprint, which can be extended with additional components. To further provide scalable processing, a suggested improvement is to automate the back end scaling by coupling it with an input queue length monitoring. This would allow to scale the back end up or down in response to changes in observed load. One interesting further development path is to port SIA to a distributed streaming environment such as Flink [19] or Spark [20]. These systems reduce the overhead of the message bus at the expense of more complex stream processing and result aggregation. While many of the existing components could be reused, some engineering effort would need to be spent on implementing a fault tolerant aggregation, integrating the potentially unreliable corpus adapters.

To encourage further discussion, the source of our current solution is freely available under an Apache 2.0 license at https://github.com/Erechtheus/sia, along with detailed guides on how to run and deploy the system.

Abbreviations

NER: Named entity recognition; SIA: Scalable interoperable annotation server; TIPS: Technical interoperability and performance of annotation servers.

Authors' contributions

JK and PT equally contributed to the implementation of SIA and to writing the manuscript. LH and RR conducted the scalability experiments, and contributed to the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability and requirements

Project name: SIA: Scalable Interoperable Annotation Server. Project home page: https://github.com/Erechtheus/sia. Operating system(s): Platform independent. Programming language: Java. Other requirements: Java 1.8 or higher. License: Apache License, Version 2.0

Funding

This research was partially supported by the German Federal Ministry of Economics and Energy (BMWi) through the projects MACSS (01MD16011F), SD4M (01MD15007B) and by the German Federal Ministry of Education and Research (BMBF) through the project BBDC (01IS14013E).

Received: 17 July 2018 Accepted: 5 December 2018
Published online: 14 December 2018

References

1. Hunter L, Cohen KB (2006) Biomedical language processing: what's beyond pubmed? Mol Cell 21(5):589–594. https://doi.org/10.1016/j.molcel.2006.02.012
2. Rheinländer A, Lehmann M, Kunkel A, Meier J, Leser U (2016) Potential and pitfalls of domain-specific information extraction at web scale. In: Proceedings of the 2016 international conference on management of data, pp 759–771. https://doi.org/10.1145/2882903.2903736
3. Thomas P, Starlinger J, Leser U (2013) Experiences from developing the domain-specific entity search engine GeneView. In: Proceedings of Datenbanksysteme Für Business, Technologie und Web, pp 225–239
4. Comeau DC, Doğan RI, Ciccarese P, Cohen KB, Krallinger M, Leitner F, Lu Z, Peng Y, Rinaldi F, Torii M et al (2013) Bioc: a minimalist approach to interoperability for biomedical text processing. Database 18:bat064
5. Pérez-Pérez M, Pérez-Rodríguez G, Blanco-Míguez A, Fdez-Riverola F, Valencia A, Krallinger M, Lourenco A (2017) Benchmarking biomedical text mining web servers at BioCreative V.5: the technical interoperability and performance of annotation servers—TIPS track. In: Proceedings of the BioCreative V.5 challenge evaluation workshop, pp 12–21
6. Thomas P, Rocktäschel T, Hakenberg J, Lichtblau Y, Leser U (2016) SETH detects and normalizes genetic variants in text. Bioinformatics 32(18):2883–2885. https://doi.org/10.1093/bioinformatics/btw234
7. Hohpe G, Woolf B (2002) Enterprise integration patterns. In: 9th conference on pattern language of programs, pp 1–9
8. RabbitMQ www.rabbitmq.com
9. Kleene SC (1956) Representation of events in nerve nets and finite automata. In Shannon CE, McCarthy J (eds) Automata studies (AM-34). Princeton University Press, Princeton, pp 3–42. https://doi.org/10.1515/9781400882618-002
10. Ambros V, Bartel B, Bartel DP, Burge CB, Carrington JC, Chen X, Dreyfuss G, Eddy SR, Griffiths-Jones S, Marshall M, Matzke M, Ruvkun G, Tuschl T (2003) A uniform system for microRNA annotation. RNA 9(3):277–279
11. Aho AV, Corasick MJ (1975) Efficient string matching: an aid to bibliographic search. Commun. ACM 18(6):333–340. https://doi.org/10.1145/360825.360855
12. Bodenreider O (2004) The unified medical language system (UMLS): integrating biomedical terminology. Nucleic Acids Res 32(Database issue):267–270
13. Leaman R, Islamaj Doğan R, Lu Z (2013) DNorm: disease name normalization with pairwise learning to rank. Bioinformatics 29(22):2909–2917
14. Rocktäschel T, Weidlich M, Leser U (2012) ChemSpot: a hybrid system for chemical named entity recognition. Bioinformatics 28(12):1633–1640
15. Gerner M, Nenadic G, Bergman CM (2010) Linnaeus: a species name identification system for biomedical literature. BMC Bioinform. 11(1):85. https://doi.org/10.1186/1471-2105-11-85
16. Leaman R, Gonzalez G (2008) Banner: an executable survey of advances in biomedical named entity recognition. In: Pacific symposium on biocomputing, World Scientific, pp 652–663
17. The national center for biotechnology information. ftp://ftp.ncbi.nlm.nih.gov. Accessed 19 Nov 2018
18. Kirschnick J, Alcaraz Calero JM, Goldsack P, Farrell A, Guijarro J, Loughran S, Edwards N, Wilcock L (2012) Towards an architecture for deploying elastic services in the cloud. Softw Pract Exp 42(4):395–408. https://doi.org/10.1002/spe.1090
19. Alexandrov A, Bergmann R, Ewen S, Freytag J-C, Hueske F, Heise A, Kao O, Leich M, Leser U, Markl V et al (2014) The stratosphere platform for big data analytics. VLDB J 23(6):939–964
20. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: Proceedings of the 2Nd USENIX conference on hot topics in cloud computing, Berkeley, USA, pp 10–10