SparkKG-ML: A Library to Facilitate End–to–End Large–Scale Machine Learning Over Knowledge Graphs in Python

Bedirhan Gergin and Charalampos Chelmis(✉)

University at Albany, SUNY, Albany, NY 12222, USA
{bgergin,cchelmis}@albany.edu

Abstract. This paper presents SparkKG–ML, the first open–source library for Machine Learning at scale over semantic data stored in Knowledge Graphs directly in Python. SparkKG–ML serves as a bridge between (i) the Semantic Web data model, (ii) the distributed computing capabilities of Apache Spark, and (iii) the Python ecosystem. By harnessing the flexibility of Python and the scalability of Spark, SparkKG–ML reduces the barriers for Data Scientists and Machine Learning researchers to work with semantic data, and for Semantic Web experts to develop Machine Learning models.

Resource Type: Software
Repository: https://github.com/IDIASLab/SparkKG-ML
License: Apache License 2.0

Keywords: Community–shared software framework · Distributed Computing · RDF · PySpark

1 Introduction

Knowledge graphs (KGs) and Linked Open Data (LOD) have become popular across fields [2,13]. Both provide convenient access to web–based knowledge, while storing and formalizing domain–specific information [17]. By analyzing KGs, scientists can identify patterns, connections, and dependencies across different data sources [13], and perhaps more importantly, infer new knowledge from given facts [16]. The capability to derive new information from existing knowledge is particularly useful in applications including question answering, recommendation systems, and expert systems.

As the popularity of KGs increases, so does their size. KGs such as DBpedia [1], which encompasses knowledge from 111 different language editions of Wikipedia, or TweetsKB [8], which comprises a collection of billions of tweet–related information spanning more than 9 years, are impossible to process with a single computer. Distributed computing offers a viable solution to this challenge by leveraging the combined capabilities of multiple servers within a cluster [7].

G. Demartini et al. (Eds.): ISWC 2024, LNCS 15233, pp. 3–19, 2025.
https://doi.org/10.1007/978-3-031-77847-6_1

Numerous distributed computing frameworks, such as Apache Spark [19], have therefore been developed and widely adopted. The Apache Spark ecosystem in particular, includes libraries for data processing and machine learning pipelines (e.g., MLlib [11] and BigDL [6]). However, there is a disconnect between Spark's native data representation and the data model of KGs. Specifically, RDF data is modeled as a graph and represented as subject, predicate, object triples, as illustrated in Fig. 1(a). Instead, Spark stores and processes tabular data using Spark DataFrames[1]. Converting between the two representations is neither trivial nor straightforward since semantic data obtained by issuing SPARQL queries (e.g., as shown in Fig. 1(b)) result in repetitive rows, as shown in Fig. 1(c), where fat amount appears multiple times for each recipe that has more than one ingredient).

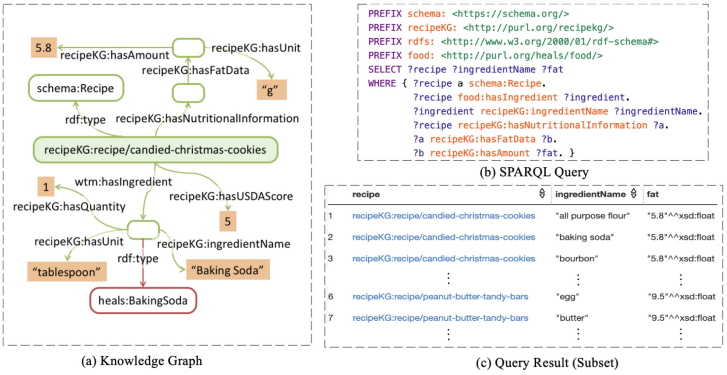


Fig. 1. (a) Subset recipe data in RDF (from RecipeKG [4]). (b) Sample SPARQL query. (c) Corresponding query result in Apache Jena Fuseki Server.

Systems such as SANSA Stack [7,10] have been proposed to facilitate large–scale distributed RDF data processing. However they require knowledge of Scala, which although popular for big data analytics, is not as popular for data science, and even less so among Semantic Web users. In contrast, Python dominates the programming language landscape[2]. Libraries, such as PySpark[3], make the functionality of Spark available in Python with the end goal of increasing the adoption of distributed computing. Other existing libraries (e.g., RDFLib[4], SPARQLWrapper[5]) are primarily concerned with RDF data access in Python, leaving other key steps (e.g., feature transformation between data models) in the pipeline of developing machine learning models over semantic data to the user.

[1] https://spark.apache.org/docs/latest/sql-programming-guide.html.
[2] https://www.tiobe.com/tiobe-index/.
[3] https://spark.apache.org/docs/latest/api/python/.
[4] https://github.com/RDFLib/rdflib.
[5] https://github.com/RDFLib/sparqlwrapper.

This paper introduces SparkKG–ML, a new library that aims to facilitate, for the first time, end–to–end Python pipelines for machine learning with Spark over semantic data stored in KGs. SparkKG–ML serves as a bridge between the semantic web data model, the distributed computing capabilities of Apache Spark, and the Python ecosystem. By harnessing the flexibility of Python and the scalability of Spark, SparkKG–ML reduces the barriers for data scientists and machine learning researchers to work with semantic data, and for semantic web experts to develop machine learning models.

The remainder of this paper is structured as follows: Sect. 2 summarizes Related Work. Section 3 introduces the proposed SparkKG–ML library. Section 4 lists its key properties. Section 5 offers a detailed experimental evaluation of SparkKG–ML. Finally, Sect. 6 presents our availability, reusability, and sustainability plan. Section 7 summarizes our conclusions and future work.

2 Related Work

Frameworks such as Apache Spark [19] and Apache Flink[6] facilitate large–scale distributed data processing. Spark's Machine Learning library (i.e., MLlib) [11] simplifies the scaling of common machine learning algorithms, while BigDL [6] adds deep learning functionalities to Spark workflows so as to analyze data on the same cluster where the data are stored. However, using Spark to analyze semantic data stored in KGs has proven challenging, mainly due to the differences in data representation between Spark and the Semantic Web. SANSA [10] facilitates native RDF KG–based ML, but currently lacks support for handling multi-modal features and providing comprehensive explainable machine learning pipelines. DistRDF2ML [7] introduced modules into SANSA to transform RDF data into ML–ready fixed–length numeric feature vectors. However, both SANSA and DistRDF2ML are Scala–based. In contrast, data scientists and researchers are primarily working in Python, whereas Semantic Web experts are accustomed to Semantic Web technologies. It is therefore not surprising that neither has been widely adopted.

On the other hand, Python provides a wealth of resources for every stage of ML pipelines, including popular libraries, such as Scikit–learn[7]. Python libraries for Semantic Web data, include RDFLib, SPARQLWrapper (for remote query execution) and kgextension [3] (for augmentation of existing datasets with information obtained from LOD). The main limitation of such libraries is that they only address data access, KG creation, and serialization. Currently, there is no library that seamlessly integrates both these aspects and supports complete machine learning pipelines in the Python ecosystem for Semantic Web data. This work addresses this challenge by bridging, for the first time, the Python ecosystem, Apache Spark, and KGs.

[6] https://flink.apache.org.
[7] https://scikit-learn.org/stable/.

3 SparkKG–ML

The process of implementing a ML pipeline over a cluster using existing Python infrastructure is illustrated in Fig. 1a. Initially, a library like RDFLib (or SPARQLWrapper) must be used to query a local RDF dataset (similarly remote SPARQL endpoint). Once the data is obtained, it must be subsequently transformed so it can be stored into an appropriate structure, such as a Pandas DataFrame[8]. The next step involves converting it into a Spark DataFrame for use in PySpark. Such transformations entail addressing certain challenges, including the need to understand all data types or infer them if unkown. Once transformed into a Spark DataFrame, the identification of feature properties is crucial for determining the appropriate vectorization method to digitize features. For feature selection or data augmentation methods to be subsequently applied, data must exit the Spark environment, resulting in unnecessary data transfers and transformations. Vectorization methods are applied manually to every feature in preparation for model training using Spark MLlib. Finally, the outputs of ML models must again be transformed into a Pandas DataFrame, and/or serialized in RDF using RDFLib.

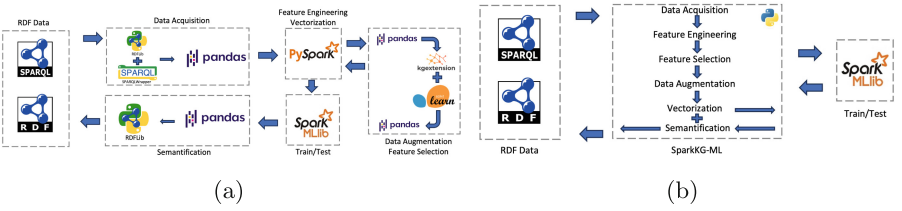


Fig. 2. (a) ML pipeline on semantic data with distributing computing in Python. (b) ML pipeline on semantic data with SparkKG–ML.

Our *open source* library, SparkKG–ML, simplifies this manual, labor intensive and data–transfer and transformation costly process, as shown in Fig. 2b. It comprises 6 modules (described in the following subsections) that align with typical data science workflows [7,15]. First, users can query RDF data (stored either locally or in a remote SPARQL endpoint) using the Data Acquisition module. The result is stored as a Spark DataFrame, upon which users may employ the Feature Engineering module. They may choose to utilize the Feature Selection or Data Augmentation modules for further enhancement, and then directly vectorize the data through the Vectorization module. Finally, after training a ML model in Spark MLlib, users can serialize model outputs using the Semantification module.

[8] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html.

3.1 Data Acquisition

This module is designed to transform RDF data into Spark's tabular format. It allows for data to be acquired from a SPARQL endpoint or a local RDF file, given a SPARQL query. Prior to transformation into a Spark DataFrame, null values are eliminated through data handling functions, including Null Drop and Null Replacement. The module provides options for users to handle null values, and to customize the data according to their needs. For instance, one can choose to drop rows or columns after a certain percentage threshold of null values is exceeded or replace null values with custom values.

Figure 3a illustrates the the process of acquiring data from RecipeKG, with the resulting Spark Dataframe shown in Fig. 3b. In this case the Data Acquisition module retrieves recipes along with their fat amounts and ingredients using the query shown in Fig. 1b. Since each recipe comprises multiple ingredients, it is repeated over multiple rows in the resulting table.

```
# Import the required module
from sparkkgml.data_acquisition import DataAcquisition

# Create an instance of DataAcquisition
dataAcquisitionObject=DataAcquisition()

# Specify the SPARQL endpoint and query
endpoint = "https://recipekg.arcc.albany.edu/RecipeKG"
query ="""  ...  """

# Retrieve the data as a Spark DataFrame
spark_df = dataAcquisitionObject.getDataFrame(endpoint=
    endpoint, query=query)
```

(a)

recipe	ingredient	fat
candied-chri...	flour	5.8
candied-chri...	baking soda	5.8
candied-chri...	bourbon	5.8
candied-chri...	brown sugar	5.8
candied-chri...	butter	5.8
peanut-butte...	egg	9.5
peanut-butte...	butter	9.5
peanut-butte...	chocolate	9.5
peanut-butte...	baking powder	9.5
the-best-oat...	cinnamon	7.6

(b)

Fig. 3. (a) Data Acquisition Module usage example. (b) Sample output of the Data Acquisition Module.

3.2 Feature Engineering

To be compatible with ML models in Spark, all features need to be converted into fixed–length numeric feature vectors. Thus, non–numeric features require conversion into numeric representations. The choice of method depends on the specific characteristics of each feature, and this module is responsible for extracting such characteristics. In addition, as depicted in Figs. 1 and 3b, the nature of triple representation leads to recipes appearing multiple times in the output, with each instance corresponding to its respective information. A collapsed DataFrame is created, similar to [7], where each sample is represented by a single row containing all relevant features. A sample output of this module when the Spark DataFrame shown in Fig. 3b is provided as input, is shown in Fig. 5a.

In addition to generating a collapsed DataFrame, the Feature Engineering module identifies the "ingredients" column as a non–categorical list of strings, and the "fat" column as non–categorical float values according to metadata

explained next. This information is stored internally to assist the Vectorization module (Subsect. 3.5) in selecting the appropriate method to digitize these columns.

Beyond this illustrative example, the list of feature metadata retained for use in the vectorization process is listed below:

- *datatype:* The data type of the feature column.
- *numberDistinctValues:* The number of distinct values in the feature column.
- *isListOfEntries:* Flag indicating if the feature is a list of entries.
- *isCategorical:* Flag indicating if the feature is categorical. The ratio of distinct values and overall dataset size is used to compute this attribute [7].
- *featureType:* Combine features based on whether they consist of a list or a single value, categorical or non-categorical, and data type.

```
# Import the required module
from sparkkgml.feature_engineering import FeatureEngineering
from sparkkgml.vectorization import Vectorization

# Create an instance of FeatureEngineering
featureEngineeringObject=FeatureEngineering()

# Call the getFeatures function
df2,features=featureEngineeringObject.getFeatures(spark_df)

# Create an instance of Vectorization
vectorizationObject=Vectorization()

# Call vectorize function, digitaze all the columns
digitized_df=vectorizationObject.vectorize(df2,features)
```

(a)

```
# Import the module
from sparkkgml.semantification import Semantification

# Create Semantification instance
semantificationObject=Semantification()

# Prepare the necessary information for semantification
semantificationObject.semantify(df, entityid='uri', label='
    label', prediction='prediction', dest='experiment.ttl')
```

(b)

Fig. 4. (a) Feature Engineering and Vectorization Modules usage example. (b) Semantification Module usage example.

3.3 Data Augmentation

This module utilizes the kgextension library [3] to augment a given KG with additional features extracted from public KGs. Preconfigured connections to DBpedia [1] and Wikidata [18] are included, although custom SPARQL endpoints or local RDF repositories can also be used. For example, using the RecipeKG example, one can extract features such as cuisines and culture from DBpedia, given recipe names. Acquired external data are added to the existing Spark DataFrame as new new columns.

3.4 Feature Selection

Feature selection focuses on identifying informative attributes, while discarding redundant ones, to enhance model accuracy, reduce computation time, and even promote a deeper understanding of the learning model [12]. Beyond facilitating large–scale Feature selection through Spark's MLlib, the Feature Selection module implements popular feature selection methods, including forward sequential feature selection and correlation feature selection, which are not currently provided by Spark.

recipe	ingredients	fat
candied-chri...	[baking soda, egg, b...	5.8
peanut-butte...	[egg, butter, chocol...	9.5
best-oatmeal...	[cinnamon, egg, suga...	7.6
alfredo-blue...	[salt, egg, pasta, b...	5.2
millie-pasqu...	[lemon, flour, salt,...	4.3

(a)

entity	features	label
candied-chri...	[0.01, 1.34, 6...	5
peanut-butte...	[0.03, 6.34, 7...	6
best-oatmeal...	[0.01, 8.34, 3...	1
alfredo-blue...	[0.05, 3.34, 2...	5
millie-pasqu...	[0.61, 9.34, 1...	4

(b)

Fig. 5. (a) Sample output of the Feature Engineering Module (b) Sample output after using the Vectorization Module and Vector Assembler.

3.5 Vectorization

This module produces a ML–ready DataFrame by transforming all features according to their feature type into numeric representations, and utilizing the Vector Assembler function provided by Apache Spark as follows:

- *Single Categorical String:* For columns containing a single categorical string, either string indexing or hashing is performed based on user inputs.
- *List of Categorical Strings:* When dealing with columns consisting of a list of categorical strings, the function explodes the list and applies string indexing or hashing based on the configured strategy. For instance, in RecipeKG [5], recipes are associated with (multiple) categories (e.g., Seafood, Dessert).
- *Single Non–Categorical String:* Columns with a single non-categorical string are processed by applying Word2Vec embedding after tokenization. Optional stop word removal can also be performed.
- *List of Non–Categorical Strings:* In the case of columns containing a list of non–categorical strings (e.g., the ingredients of a recipe in Fig. 5a), the list elements are combined, tokenized, and optionally, stop words are removed. Embeddings are then calculated using Word2Vec.
- *Numeric Type:* For columns of numeric types (i.e., integer, long, float, double), both single and list types are handled by either joining or exploding the values.
- *Boolean Type:* Columns of boolean type are cast to integers (0 or 1).
- *Date and Time datatypes:* In its current implementation, SparkKG–ML treats these as string literals.

Once all features have been obtained, they are combined into a single row of type Array of Doubles using Vector Assembler. This finalizes the process, enabling ML models to be trained through Apache Spark's MLLib [11] or BigDL [6] libraries. We illustrate the process using a classification task, in which recipes are assigned health scores [4]. Figure 4a shows the simplicity of invoking the Vectorization module, whereas Fig. 5b shows the resulting DataFrame with feature vectors and health scores as labels obtained from the Vector Assembler after being vectorized.

3.6 Semantification

The Semantification module completes the end–to–end process of a ML workflow by transforming ML results into RDF data, as shown in Fig. 4b. Specifically,

Machine Learning models typically generate results that include predictions, associated data, model identifiers, and other relevant information. This module processes the final output dataframe by extracting entities and prediction results from the columns, and subsequently serializes this information into RDF format, structured according to a user–defined ontology. The process of converting and storing ML results in standardized RDF format simplifies the process of querying and visualizing them, and has the potential to enhance their interoperability, integration with other knowledge graphs and semantic ecosystems, and sharing with other users.

4 SparkKG–ML Properties

In this section, we summarize the key properties of our library.

Distributed Computing: A key challenge in implementing big data and machine learning methods at scale lies in the limitations of local compute resources. SparkKG–ML facilitates the use of Apache Spark's distributed computing capabilities for large–scale processing of semantic data.

Accessibility: Frameworks and libraries dedicated to semantic data and ML pipelines (e.g., SANSA Stack [10]) appear to be outdated, as evident by the lack of updates or complete inactivity in their GitHub repositories. In contrast, our library is interoperable with the most recent versions of Python and PySpark (at the time of submission, versions Python $\geq$ 3.8, and PySpark $\geq$ 3.2). Its entire code base is openly accessible, and its installation boils down to a single command line via the Python Package Index. Comprehensive documentation, including a step–by–step tutorial, is available on readthedocs. The corresponding links are provided in Sect. 6.

Encapsulation: Working with distributed computing infrastructures demands significant time investment when building an ML pipeline for semantic data. This challenge is shared between semantic web experts and data scientists alike. SparkKG–ML streamlines and consolidates essential ML pipeline preparation steps. Specifically, the proposed library encapsulates various procedures into a few lines of Python code (or even a single function), automating a number of processes, as highlighted in Sect. 3, and illustrated by Fig. 2a and 2b.

Ease of Use: In addition to condensing numerous steps into simple function calls, the proposed library leverages the inherent simplicity, conciseness, and straightforward syntax of the Python programming language. Specifically, it capitalizes on Python's rich collection of built-in compound objects, which is a benefit for enhancing code readability, and simplifying complex tasks. The rich ecosystem built around Python further enables the integration of free external libraries into a machine learning pipeline once semantic data have been transformed into the tabular format required by such libraries.

Generality and Extensibility: Our experimental evaluation (Sect. 5) demonstrates the versatility and generality of the proposed library for a variety of data

types. Its modules are designed to provide users with readily available tools to enhance their ML pipelines over semantic data from one end to the other. Nevertheless, since one cannot possibly account for all requirements, the library is open-sourced, facilitating, and even encouraging extensibility (e.g., with other feature selection methods).

Scalability: Our experimental evaluation (Sect. 5) demonstrates the scalability of the proposed library as a function of both dataset size, result set size and query complexity, all important factors when operating with semantic data.

5 Experiments

We investigate various parameters, including processing power, Spark cluster configuration, SPARQL complexity, dataset size, and result set size, analyzing their influence on overall runtime performance. This analysis is conducted in comparison with the existing framework DistRDF2ML [7]. In the second part, we delve into the performance of our library's additional functionality. Both runtime efficiency and overall performance metrics are reported.

Setup

We utilized Databricks Community Edition[9] with 15.3 GB Memory and 2 Cores for experiments involving comparisons with DistRDF2ML [7]. To set up DistRDF2ML [7] on Databricks, we followed the instructions provided through SANSA Github[10]. For the experiments assessing only SparkKG-ML performance, we utilized Oracle Cloud Infrastructure[11] and its Data Science and Data Flow services, providing up to 64 cores per executor, and up to 512 GB of memory in total. Our Databricks configuration used Python 3.9, PySpark 3.5, and Scala 2.12, while our Oracle Cloud setting used Python 3.8 and PySpark 3.2. Each experiment was run 10 times, and the average performance is reported. All results and codes are available on GitHub[12] for reproducibility.

Datasets

In our experimental evaluation, we use two real–world datasets, namely (i) Linked Movie Database (LMDB) [9] – An openly accessible knowledge graph of facts about 38,000 movies (including title, runtime, actors, genres, producers, and country of origin), (ii) IMDb Non–Commercial Dataset[13], and synthetic datasets[14], initially created for the evaluation of DistRDF2ML [7].

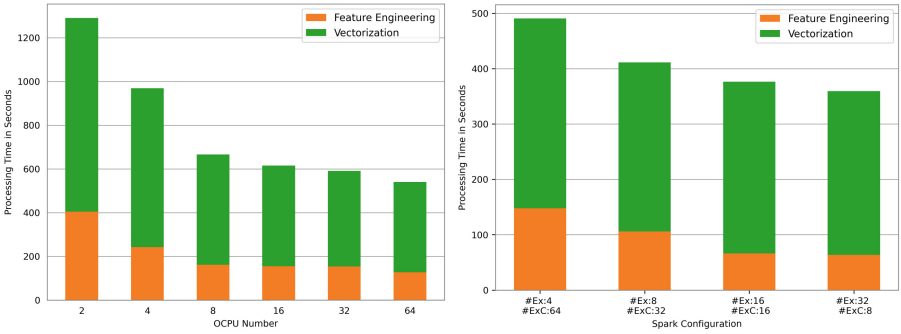
[9] https://www.databricks.com.
[10] https://github.com/SANSA-Stack/SANSA-Databricks.
[11] https://www.oracle.com/cloud/.
[12] https://github.com/IDIASLab/SparkKG-ML.
[13] https://developer.imdb.com/non-commercial-datasets/.
[14] https://github.com/SANSA-Stack/SANSA-Stack/releases/tag/v0.8.1_DistRDF2ML.

We use LMDB (small dataset) for fair comparison with DistRDF2ML [7]. We use IMDb (large dataset) to demonstrate the scalability of SparkKG-ML. The dataset comprises facts (i.e., title, year, runtime, and genres) for $10,263,447$ movies. In synthetic datasets, the dataset size is exponentially increased (controlled by the number of movies), while keeping the resultset size and feature density unchanged.

5.1 Performance

Processing Time vs Processing Power

We exclude runtime of Data Acquisition module in this section and the next section to focus on the processing power of the machine while using IMDb Non–Commercial Dataset to extract and vectorize the movies and relevant features, as data acquisition would include the time to transfer the data. As demonstrated in Fig. 6a, an increase in processing power resulted in a substantial decrease in runtime. Moreover, we will delve into how leveraging Spark's parallelism can further enhance performance, including optimizing the executor number and providing parallelism to achieve even greater efficiency in the next section.



(a) (b)

Fig. 6. (a) Processing time as a function of number of CPUs, for a single executor. (b) Processing time as a function of number of executors, and number of executor cores (abbreviated as #Ex and #ExC, accordingly).

Processing Time vs Spark Setup

As mentioned earlier, leveraging the distributed computing capabilities of Spark is important for large-scale processing of semantic data. Next, we explore the influence of different Spark configurations on the execution time of SparkKG-ML, aiming to understand how Spark's parallelism and distributed computing capabilities impact overall runtime performance. The results in Fig. 6b reveal

that configurations with a higher number of executors, each of which has less CPUs, achieve better performance in our workload for the IMDb dataset, with the optimal setting being 32 executors with 8 cores per executor.

It's essential to emphasize that these findings are specific to the characteristics of the tested workload. Spark job performance is subject to variation based on factors such as data size, job complexity, and computation requirements. Nevertheless, these observations underscore the critical importance of striking a delicate balance between executor size and number. This highlights the need for configurations that leverage the advantages of parallelization while avoiding the drawbacks of insufficient core allocations within each executor.

Processing Time vs SPARQL Query Complexity

As highlighted in [7], the complexity of SPARQL queries can fluctuate based on factors including the count of projection variables, the volume of features, the depth of traversal required to access a particular projection variable or feature, and the type of feature. We therefore experiment with 3 progressively complex queries (illustrated in Fig. 7a) as follows:

- $\mathbf{Q_1}$ (small): Select only one variable per movie, namely its URI.
- $\mathbf{Q_2}$ (medium): In addition to URI, retrieve the title for each movie.
- $\mathbf{Q_3}$ (big): In addition to URI and title, retrieve the release date, runtime, actors, and genre of each movie.
- Queries $\mathbf{Q_4}$ and $\mathbf{Q_5}$ refer to the execution of $\mathbf{Q_3}$ on progressively larger dataset and resultset sizes, accordingly.

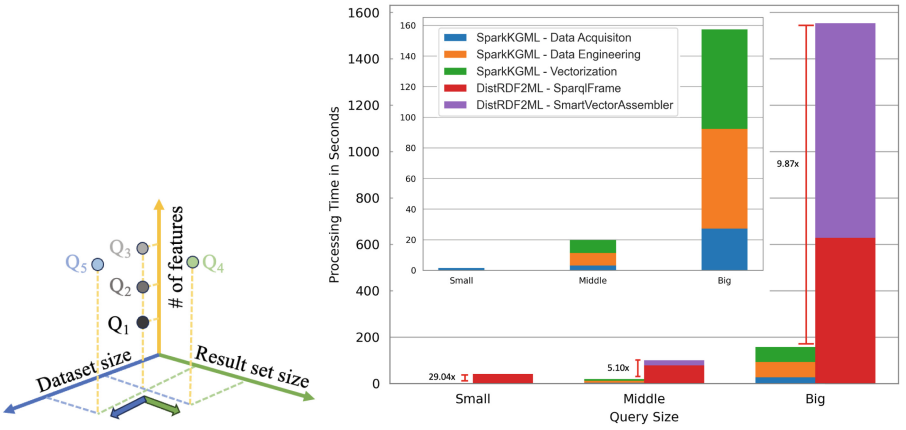
Note that $\mathbf{Q_1}$ doesn't necessitate additional processing for feature engineering and vectorization.

As demonstrated in Fig. 7b, SparkKG-ML consistently outperforms DistRDF2ML with respect to time needed to perform data acquisition, data engineering, and vectorization, for all three types of queries. Specifically, SparkKG-ML is 29.04X faster with respect to the small query (i.e., $\mathbf{Q_1}$), and 9.87X times faster with respect to the big query (i.e., $\mathbf{Q_3}$). The inset in Fig. 7b highlights the performance of SparkKG-ML, which is otherwise hidden because of the scale of magnitude difference in time required by DistRDF2ML in the main plot.

We examine the affect of increasing the dataset and result set size while keeping the query unchanged in the following two subsections.

Processing Time vs Result Set Size

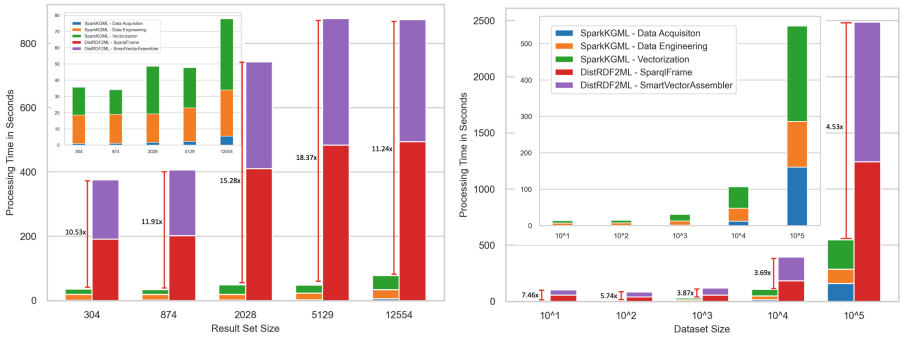
In this section, we consider a simple binary classification task, predicting the genre of movies based on features acquired using $\mathbf{Q_3}$, but with a varying result set size. Initially, we started with two genres encompassing 304 movies. Subsequently, we incorporated a greater number of entities and further increased the result set size by modifying the target genres in the dataset.



(a) (b)

Fig. 7. (a) Query complexity as a function of number of features, and dataset and result set size. (b) Processing time as query complexity increases.

The inset in Fig. 8a better shows the performance of SparkKG-ML. Evidently, SparkKG-ML remains consistently effective as the result set size increases. Furthermore, SparkKG-ML significantly outperforms DistRDF2ML across all result set sizes. It is noteworthy that the speedup achieved increased from 10.53X (for the smallest result set size, i.e., leftmost bar) to 18.37X (for a result set size of 5129).



(a) (b)

Fig. 8. (a) Processing time as a function of result set size. (b) Processing time as a function of dataset size (number of movies).

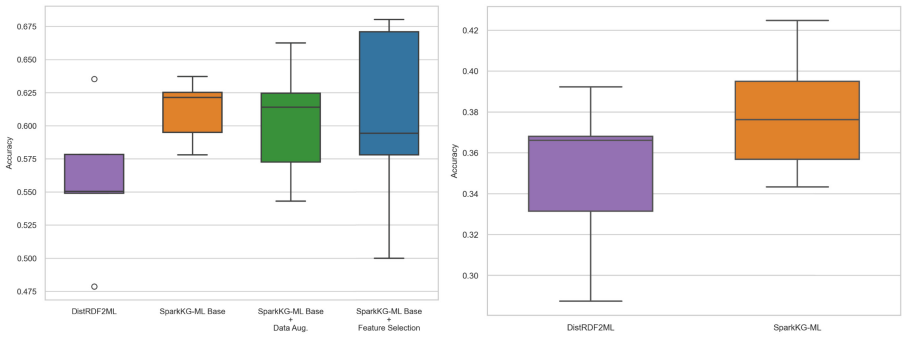
Processing Time vs Dataset Size

In this section, we utilized synthetic datasets initially created for the evaluation process of DistRDF2ML [7]. These synthetic datasets were designed to achieve exponential growth in data by incrementally increasing the size of movie numbers from 10^1 up to 10^5.

Figure 8b demonstrates the scalability of SparkKG-ML, even for exponentially increasing data sizes. It additionally shows the competitive advantage of SparkKG-ML compared to DistRDF2ML across dataset sizes. Specifically, SparkKG-ML runs 7.46X faster in the smallest dataset size and 4.53X faster in the biggest dataset size.

5.2 Functionality

In this section, we assess the additional capabilities of SparkKG-ML. Specifically, we consider a two-classification setting for predicting the genre of movies. In the first scenario, two genres are taken into account (namely Spy & Superhero). In the second scenario, the movies are divided into four categories (i.e., Spy, Superhero, Parody, and Zombie). It is important to note that our objective here is not to derive the most accurate model but rather to showcase the capabilities of SparkKG-ML's compared to DistRDF2ML.



(a) (b)

Fig. 9. (a) Binary classification accuracy comparison. (b) Multiclass classification accuracy comparison.

For fair comparison, we created identical pipelines by using the same features, classification models, and evaluation methods. In addition, we integrated the Data Augmentation module from SparkKG-ML into our pipeline, enriching our dataset with 10 additional movie features retrieved from the DBpedia endpoint. These features encompass boolean values indicating whether a movie is related to or possesses a specific characteristic. Lastly, we also incorporated our Feature

Selection module, which is used to reduce the number of features by retaining only those features that improve accuracy.

Focusing on binary classification, Fig. 9a illustrates that, on average, SparkKG-ML achieves a 10% improvement in accuracy over DistRDF2ML. The same trend can be observed for multiclass classification, as shown in Fig. 9b, with an average accuracy improvement of 2.7%.

Figure 9a further illustrates the potential benefit of SparkKG-ML's Data Augmentation Module. Specifically, the module was used without any intention of optimizing classification accuracy. Therefore 10 features out of those retrieved by the module were randomly selected. These include features such as "2010s_English-language_films", "American_black-and-white_films" Although average accuracy is comparable to the one achieved by the base model, certain values surpass the maximum achieved by the base model. Once could envision scenarios where users would optimize for accuracy by either carefully selecting a subset of features retrieved by the Data Augmentation Module, or by automating the process completely, using the Feature Selection Module in succession to the Data Augmentation Module.

Finally, Fig. 9a shows that applying Sequential Feature Selection (i.e., using the Feature Selection module) results in a performance improvement of 5% for approximately half of the test instances.

5.3 Ease of Use

We concluded our analysis with a comparison of the lines of code required to achieve the same end–goal (i.e., to train a machine learning model using semantic data) directly in Python (i.e., without the use of SparkKG-ML) or in Scala (i.e., using SANSA Stack with DistRDF2ML). Our code is available on Github[15].

Results shows that the complete process can be implemented in Python in 50 lines of code (excluding library imports), as compared to only 15 lines when using SparkKG-ML. On the other hand, one can achieve the same result using DistRDF2ML, but in Scala (as opposed to Python). Thus, the advantage of using SparkKG-ML with respect to a Python implementation is in abstracting and simplifying the process. With respect to a Scala implementation, the benefit comes from the ability to operate directly in Python, with the additional benefit of scalability and time performance improvements (as discussed in Sect. 5).

6 Availability, Reusability and Sustainability

Its source code is available at Github[16] including the experiments for this paper, while the library itself can be installed via the Python Package Index[17]. A comprehensive documentation, including a step–by–step tutorial, is accessible

[15] https://github.com/IDIASLab/SparkKG-ML/tree/main/experiments.
[16] https://github.com/IDIASLab/SparkKG-ML.
[17] https://pypi.org/project/sparkkgml/.

through readthedocs[18]. To encourage collaboration and ensure sustainability, we actively monitor the repository's issue tracker for requests, reported errors, or any specific concerns related to the library.

7 Conclusion and Future Work

This paper introduced an open–source Python library designed to streamline machine learning tasks using Spark for semantic web and knowledge graph data. To achieve this goal the proposed library, SparkKG-ML, solves several engineering challenges including serializing SPARQL query results into a Spark Dataframe, developing custom implementations or extensions to existing libraries to support common RDF serialization formats (i.e., Turtle, RDF/XML, and N-Triples), preforming datatype tailored cleaning and imputation, and integrating other existing libraries. In addition to showcasing the SparkKG-ML's capabilities, and presenting various use cases demonstrating its effective application, the paper also conducted a comparative analysis with the status quo.

Since its public release in Oct of 2023, SparkKG-ML has been downloaded more than 826 times in PyPi[19], a promising start and an indicator of the general interest in its capabilities. We anticipate that this paper will increase awareness about (and hopefully generate interest in) the library, and therefore help foster the creation of an active community, that will ultimately lead to its collective improvement by (i) introduction of new features, (ii) resolution of identified issues, (iii) performance improvements, and (iv) forward compatibility with Python and Pyspark updates to ensure its long term sustainability.

Despite its capabilities, SparkKG-ML also has shortcomings, some of which we summarize below. First, it does neither provide native support for certain data types, nor does it fully leverage the semantics of data stored in knowledge graphs. Hovered, our immediate future releases will support date extraction from strings (year, month, and day) and date/time arithmetic through new functions. Second, it leverages Word2Vec as a default model for encoding as it comes readily available through SparkMLlib. To facilitate use of other, more advanced, embedding methods we plan to build upon popular packages such as RDF2Vec [14]. The additional benefit of using such models will be improved representation of entities based on their relative connections to other entities in knowledge graphs (as opposed to bag–of–words representations obtained using Word2Vec. Finally, we plan to leverage pre-trained Large Language Models to enable end-users, who may be unfamiliar with Semantic Web standards, to generate SPARQL queries directly from natural language prompts, thereby democratizing access to, and analytics over, semantic data. Specifically, we envision users prompting LLMs with natural language text to construct SPARQL queries. More elaborate functions may include KG-enhanced LLM pre-training (i.e., injecting knowledge into

[18] https://sparkkgml.readthedocs.io/en/latest/index.html.

[19] The number of downloads has been computed using a Google BigQuery as described in https://packaging.python.org/en/latest/guides/analyzing-pypi-package-downloads/.

LLMs during pretraining), as well as LLM-automated ML pipelines (e.g., query data, train a model and plot the results).
Resource Availability Statement:

- Source code is available at Github[20] including the experiments for this paper.
- Library can be installed via the Python Package Index[21].
- A comprehensive documentation is accessible through readthedocs[22].

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_52
2. Belcao, M., Falzone, E., Bionda, E., Valle, E.D.: Chimera: a bridge between big data analytics and semantic technologies. In: Hotho, A., et al. (eds.) ISWC 2021. LNCS, vol. 12922, pp. 463–479. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88361-4_27
3. Bucher, T.-C., Jiang, X., Meyer, O., Waitz, S., Hertling, S., Paulheim, H.: `scikit-learn` pipelines meet knowledge graphs. In: Verborgh, R., et al. (eds.) ESWC 2021. LNCS, vol. 12739, pp. 9–14. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80418-3_2
4. Chelmis, C., Gergin, B.: A knowledge graph for semantic-driven healthiness evaluation of recipes. Semant. Web J. (2021). https://www.semantic-web-journal.net/content/knowledge-graph-semantic-driven-healthiness-evaluation-online-recipes
5. Chelmis, C., Gergin, B.: A Knowledge graph for semantic-driven healthiness evaluation of online recipes. (2022). https://doi.org/10.7910/DVN/99PNJ5
6. Dai, J.J., et al.: BigDL: a distributed deep learning framework for big data. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 50–60. SoCC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3357223.3362707
7. Draschner, C.F., Stadler, C., Bakhshandegan Moghaddam, F., Lehmann, J., Jabeen, H.: DistRDF2ML - scalable distributed in-memory machine learning pipelines for RDF knowledge graphs. In: Proceedings of the 30th ACM International Conference on Information and Knowledge Management, pp. 4465–4474. CIKM '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3459637.3481999
8. Fafalios, P., Iosifidis, V., Ntoutsi, E., Dietze, S.: TweetsKB: a public and large-scale RDF corpus of annotated tweets. In: Gangemi, A., et al. (eds.) ESWC 2018. LNCS, vol. 10843, pp. 177–190. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93417-4_12
9. Hassanzadeh, O., Consens, M.P.: Linked movie data base. In: LDOW (2009). https://api.semanticscholar.org/CorpusID:16810971
10. Lehmann, J., et al.: Distributed semantic analytics using the SANSA stack. In: International Workshop on the Semantic Web (2017)

[20] https://github.com/IDIASLab/SparkKG-ML.
[21] https://pypi.org/project/sparkkgml/.
[22] https://sparkkgml.readthedocs.io/en/latest/index.html.

11. Meng, X., et al.: MLlib: machine learning in apache spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)
12. N, T.R., Gupta, R.: Feature selection techniques and its importance in machine learning: a survey. In: 2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS), pp. 1–6 (2020). https://doi.org/10.1109/SCEECS48394.2020.189
13. Paulheim, H.: Machine learning with and for semantic web knowledge graphs. In: Reasoning Web (2018)
14. Steenwinckel, B., Vandewiele, G., Agozzino, T., Ongenae, F.: pyRDF2Vec: a python implementation and extension of RDF2Vec. In: Pesquita, C., et al. (eds.) The Semantic Web. LNCS, pp. 471–483. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33455-9_28
15. Svetashova, Y.: Ontology-enhanced machine learning: a Bosch use case of welding quality monitoring. In: Pan, J.Z., et al. (eds.) ISWC 2020. LNCS, vol. 12507, pp. 531–550. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62466-8_33
16. Tian, L., Zhou, X., Wu, Y.P., Zhou, W.T., Zhang, J.H., Zhang, T.S.: Knowledge graph and knowledge reasoning: a systematic review. J. Electron. Sci. Technol. **20**(2), 100159 (2022). https://doi.org/10.1016/j.jnlest.2022.100159
17. Tiddi, I., Schlobach, S.: Knowledge graphs as tools for explainable machine learning: a survey. Artif. Intell. **302**, 103627 (2022). https://doi.org/10.1016/j.artint.2021.103627
18. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014). https://doi.org/10.1145/2629489
19. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664