Feature-Oriented Modularization of Deep Learning APIs

Yechuan Shi

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of

Master of Computer Science

School of Computer Science
McGill University
Montréal, Québec, Canada

Oct 2022

Abstract

Deep learning libraries provide vast APIs because of the multitude of supported input data types, pre-processing operations, and neural network types and configuration options. However, developers working on one concrete application typically use only a small subset of the API at any one given time. Newcomers hence have to read through tutorials and API documentation, gathering scattered information, trying to find the API that fits their needs. This is time consuming and error prone. To remedy this, we show how we modularized the API of a popular Java DL framework *Deeplearning4j (DL4J)* according to features. Beginner developers can interactively select desired high level features, and our tool generates the subset of the DL library API corresponding to the selection. We evaluate our modularization on DL4J code samples, demonstrating an average recall of 98.9% for API *classes* and 98.0% for API *methods and constructors*. The respective precision is 19.3% and 13.8%, which represents an improvement of two orders of magnitude compared to searching through the original complete DL4J API.

Abrégé

Les bibliothèques d'apprentissage en profondeur (Deep Learning Libraries) ont souvent une interface de programmation (API) très vaste en raison de la multitude de types de données d'entrée pris en charge, d'opérations de prétraitement, de types de réseaux neuronaux et d'options de configuration. Cependant, les développeurs travaillant sur une application concrète n'utilisent généralement qu'un petit sous-ensemble de l'API à un moment donné. Les developpeurs nouveaux arrivants doivent donc lire des tutoriels et de la documentation pour connaître l'API. Ils doivent rassembler des informations éparpillées un peu partout, pour essayer de trouver l'API qui correspond à leurs besoins. Cela prend du temps et peut facilement enduire des erreurs. Pour remédier à cela, nous montrons comment nous avons modularisé l'API d'un framework Java d'apprentissage en profondeur populaire appelé *Deeplearning4J (DL4J)* par rapport au fonctionnalités qu'il offre. Les développeurs débutants peuvent sélectionner de manière interactive les fonctionnalités de haut niveau souhaitées, et notre outil génère le sous-ensemble de l'API de la bibliothèque qui correspond à cette sélection. Nous évaluons notre modularisation sur des exemples de code DL4J, démontrant un rappel moyen de 98,9% pour l'API *classes* et de 98,0% pour l'API *méthodes et constructeurs*. La précision respective est de 19,3% et 13,8%, ce qui représente une amélioration de deux ordres de grandeur par rapport à la recherche via l'API DL4J complète d'origine.

Acknowledgements

I want to thank Prof. Jörg Kienzle and Prof. Jin Guo for supervising me during my master's studies, for their guidance, patience, and insistence on high standards through each stage of this research project. Words cannot express how grateful I am for being their student.

I also want to thank my colleagues and friends for supporting my studies and my life. Last but not least, I would like to thank my families for the unconditional love and support.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Statement and Motivation

During the past two decades, the field of Artificial Intelligence (AI) has experienced dramatic growth, from a mostly academic research topic to a practical technology in commercial use. Because of this, many CS students choose to explore the field of AI during their studies nowadays. According to the AI index 2022 annual report [1], "1 in every 5 CS students who graduated with PhD degrees specialized in artificial intelligence/machine learning in 2020". With the influx of researchers in this field, the total number of AI publications doubled, growing from 162,444 in 2010 to 334,497 in 2021 according to the same report. This phenomenal growth attracts many inexperienced students or programmers into this field and they take some AI-related courses. In Udemy, a website for teaching and learning online, machine learning related courses have 7 million learners[2].

The prosperity of AI, and in particular Deep Learning (DL), caused many specialized machine learning frameworks and libraries to emerge. The APIs of DL libraries are typically vast and complex, causing beginners to experience difficulties in learning how to correctly use them.

First of all, DL algorithms can be applied to data of different types, e.g., textual data, CSV data, image or video data, and audio data. Therefore, the APIs of the DL libraries were designed to support these different formats, which causes many API elements to be defined multiple times, once for each supported data type. As a result, the API grows significantly in size.

Another obstacle for the inexperienced programmers is that decisions made during one phase of the machine learning pipeline can affect later stages. A typical machine learning architecture takes the form of a data processing pipeline, where data is read from one or several input sources, pre-processed and transformed if needed, and then used to train a neural network. The trained

[1]https://aiindex.stanford.edu/wp-content/uploads/2022/03/2022-AI-Index-Report_Master.pdf

[2]https://www.udemy.com/topic/machine-learning/

network can then be used for prediction purpose. Since ML is a pipeline with data streaming from one end to the other, the choice of API usage in one phase might affect the API that needs to be used in a later phase. For example, the data format of the input data would impact the ways of preparing the data. After the data preparation, the characteristics of the output data might influence the choice of network type.

With these two special characteristics of the machine learning libraries, unlike other libraries for traditional software development in which API usage decisions do not have cascading effects, DL libraries do require users to have good domain specific knowledge to understand the effects of their API usage decisions. For inexperienced programmers, e.g., students, it is therefore critical to assist them with their API choices.

Traditionally, programmers gain the knowledge of the library API usage by reading its documentation or by working through tutorials. Even though documentation can show the functionalities and structures of the APIs, according to *Gias et al.* [28], API documentation usually has problems because it contains bloated, scattered or tangled information. Additionally, the documentation or tutorials typically do not provide an abstraction of the library functionality at a high level. Beginners need to read or follow a significant number of tutorials to obtain a level of understanding that allows them to start writing code. What's more, beginners need to search through the entire API of the library to find what they need if they cannot obtain that information from the documentation or tutorials. Although sometimes crowd sources, e.g., Stack Overflow, can help to find high quality answers for specific questions, beginners sometimes struggle to formulate a question that will allow them to get a good answer using the search engines. Again, a service that would present a high-level overview of the functionality offered by the DL library, and modularize the corresponding API elements would greatly guide the users towards the API to be used for their desired purpose.

1.2 Thesis Contribution

The work presented in this thesis aims at helping machine learning developers, in particular newcomers that are using a DL library, to make correct use of the librarie's API.

In particular, we make the following contributions:

- By modularizing the machine learning libraries in a feature-oriented way, we present a new intuitive way of library documentation that is aligned with the data processing and machine learning pipeline. Our approach presents an interactive feature model that encodes the high-level features of the DL framework to the developer. The developer simply selects those features that they intend to use in their programming task. Our approach then generates the subset of the DL framework APIs that corresponds to the features chosen by the developer.

- To showcase the practicality of our approach, we modularized the API of the popular Java library *DeepLearning4J* (DL4J) according to features. Beginner users can interact with the DL4J feature model using the concern-oriented modelling tool TouchCORE, which ensures that the developer makes a correct feature selection. The aspect-oriented model weaver provided by TouchCORE is used to generate the API corresponding to the feature selection of the user.

- We evaluate the correctness and effectiveness of our approach to successfully retrieve API classes and methods based on feature selections. Our evaluation on the *DL4J* library demonstrates an average recall of 98.9% for API *classes* and 98.0% for API *methods and constructors*. The respective precision is 19.3% and 13.8%. While the precision rates are not high, they still represent an improvement of two orders of magnitude compared to searching through the original complete DL4J API.

1.3 Thesis Outline

The remainder of the thesis is structured as follows: Chapter 2 presents the background knowledge required to understand the rest of the thesis, especially concern-oriented reuse and the concept of concernification. It also overviews related work and existing approaches for API documentation and API recommendation. Chapter 3 explains our approach by showing how we built a feature model for DL4J, how we linked the different API elements to the features so that we can generate custom APIs from feature selections, and how it is intended to be used by a beginner developer. To show the correctness and effectiveness of our approach, chapter 4 defines recall and precision metrics and then shows how well our concernified DL4J API performs. It also discusses the threats to validity of our experiments. Finally, Chapter 5 presents a conclusion and discusses future work related to the concernification of AI libraries.

2
Background and Related Works

In this chapter, we discuss the key concepts and approaches that serve as the foundation of our work. We also describe previous literature on improving API usability, in particular centered around API documentation.

2.1 Background

2.1.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is an iterative and incremental software development process, which advocates the use of models as the primary artefact at every stage of the development life cycle. Models raise the abstraction level, allowing different stakeholders to participate in every stage of software development through different modeling formalisms and notations that are appropriate for each stage.

Among various forms of modeling choices, such as sketches, informal diagrams, etc., we decided to adopt feature models [10] and class diagram in our thesis. A feature is a concept used in software product line engineering[18]. A feature represents a distinguishing characteristic of a software product, usually visible to the customer or user of that product. Feature models encapsulate application features and their dependencies in a simple and hierarchical diagram. The features are structured in a tree where the non-root features have a relationship to their parents (i.e., mandatory or optional). Features sharing the same parent have *OR* or *XOR* (exclusive or) relationship among them. Additionally, cross-tree constraints (i.e., requires or excludes) support the definition of additional relationships.

For example, figure 2.1 shows a simple concern of a deep learning framework which supports only one type of data as network input. As deep learning involves the process of inputting data into the network, the non-root features “load data”, “specify network” and “run network” are mandatory. Under the feature “load data”, “CSV data” and “image data” have the *XOR* relationship so that they cannot be selected at the same time. The first cross-tree constraint means the feature of “operate data” requires the feature of “CSV data“ to be selected at the same time. The second cross-tree constraint means feature “operate data” and “image data” can not be chosen together. These two constraints indicate that this simple deep learning concern supports operating only CSV data.

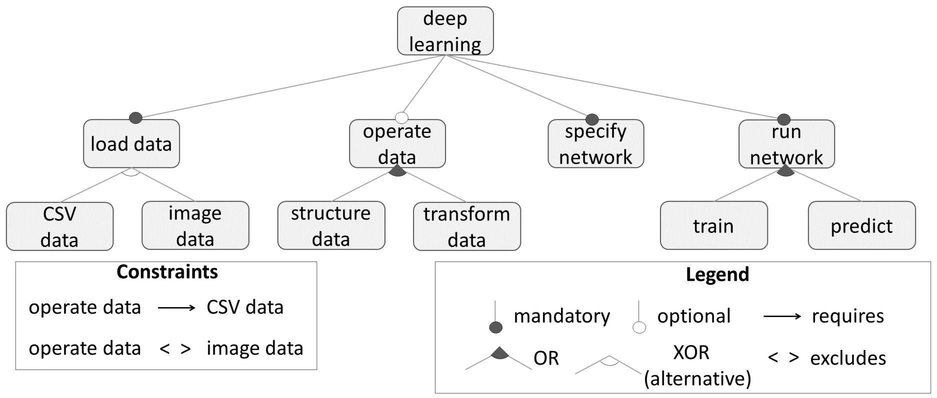


Figure 2.1: Feature Model for a Simple Deep Learning Concern

On the other hand, a class diagram works as a blueprint of the system by modeling the classes defined in the software and their inter-relationships. Class diagrams can be used by developers to effectively communicate their understanding of a domain, but also to capture the structure of a software design. As we are planning to model the API of a Java framework, and considering the fact that Java is an object-oriented language, class diagrams are a suitable means to specify the framework’s API. In this thesis, we therefore use feature models to specify the high-level functionality of the DL library and class diagrams to express its API.

2.1.2 Concern-Oriented Reuse

A software system built using object-oriented languages can be modularized with classes. Classes provide encapsulation of functionality and enable code level reuse. Moreover, a group of related classes, or components, can be packaged. For example, in Java, packages provide name spaces, and a group of packages can be put into a JAR file. This is typically how libraries are packaged, distributed and eventually reused. However, the crosscutting nature of most software concerns hinders these modularization techniques from fullfilling the design principle of Separation of Concerns (SoC), which advocates separating a software system into distinct sections with as little overlap as possible. In SoC, each module addresses a separate *concern*.

As one approach to solve the problem of cross-cutting concerns, *aspect-orientation*, separates concerns along additional dimensions. There are many kinds of concerns that are important in a software development lifecycle. Some concerns align with the domain concepts. Others come from system requirements or from the development process. According to [25], “Common dimensions of concern are data or object (leading to data abstraction) and function (leading to functional decomposition). Others include feature (both functional, such as “evaluation,” and cross-cutting, such as “persistence”), role, and configuration”. In *aspect-orientation*, concerns are usually called *aspects*. To construct the final system, all used aspects are combined by a so-called *weaver* at specified places, which are called *join points*.

Concern-Oriented Reuse (CORE) is a new reuse approach inspired by the ideas of multi-dimensional separation of concerns [25]. CORE builds upon MDE, software product lines, goal modelling, and advanced modularization techniques offered by aspect-orientation. CORE defines flexible modules, called *concerns*, to achieve a broader scale of software reuse. A concern can encapsulate any number and kind of software artefacts, from reusable models to reusable code. Furthermore, a concern in CORE encapsulates not only *one* specific way of addressing a domain of interest or software development issue, but *many* variants that can provide the same or similar functionality.

To this aim, CORE modularizes a system along features. Each concern has its feature model to reflect the available variants encapsulated within the concern. Each feature (or combination of features) can then be associated with the reusable artefacts that describe the structure or behaviour of that feature. These artefacts that are attached to features are called *realization models*. Based on a feature selection in the feature model, a specific configuration of the concern can then be obtained by composing the realization models corresponding to the feature selection.

2.1.2.1 Composition

Typically, a realization model only contains those model or code elements related to a single feature. As different features might share the same elements, there is also potential for reuse within the same concern. To support that, a realization model can extend several realization models in CORE, thus sharing the model or code elements. To generate a specific configuration of a concern based on a specific feature selection, CORE composes all realization models attached to the selected features together. The composition algorithm [1] also keep track of the pair-wise relationships between each of the composed elements in the composed model, and from which feature(s) and realization model the element originated from.

2.1.2.2 Concern Interface

CORE promotes modularity through three concern interfaces that every concern must provide [11]: *variation interface*, *customization interface*, *usage interface* (VCU).

- The *variation interface* describes the available variants of a concern, typically presented by the feature model. Also, the impact model is provided to give the users an indication of how their feature selections influence the application quality.
- The *customization interface* describes how a chosen variant can be adapted to the needs of a specific application. To allow a flexible reuse of a concern, the content in the concern cannot be completely specified since it depends on the context in which it is applied. As a result, some elements act as placeholders and need to be completed by the users. For example, to use *Observer Pattern* as a concern, the users need to point out which element in the application should play the role of its *subject* and *observer*.
- The *usage interface* describes how the application can finally access the structure and behaviour provided by the concern. For the purpose of information hiding, the details of the structure and behaviour are not accessible from the outside. Similar to the concepts of API, the usage interface exposes only the necessary elements to the users to trigger the functionality provided by the concern.

2.1.2.3 The CORE Reuse Process

While the process of building a concern is non-trivial and time-consuming, reusing an existing concern is simple, essentially involving three steps supported by the three concern interfaces:

1. Selecting the features in the variation interface which can best fulfill the user's need; the variation interface organizes the possible features as variations and their impact on goals and system qualities. the impact of choosing a feature can be specified with goal models, according to which users select the most appropriate features. Based on this selection, the CORE tool generates a generic version of the concern containing only those realization models related to the selected features.
2. Adapting the generic realization models generated in the previous step to the specific reuse context based on the customization interface; this is done by mapping partial structural and behavioural model elements of the customization interface to structure and behavioural elements in the reuse context.
3. Using the model correctly through the usage interface; for example, if the realization models of the concern take the form of class diagrams, then the usage interface would consist in public classes and public methods that can be invoked by the user.

2.1.3 Concernification

Concernification is a process of creating a *concern interface* for a reusable artefact (such as framework) by raising the level of abstraction of its APIs to the modeling level [23]. It enables reuse of existing APIs or code at the modeling level. To concernify an existing framework, the following steps are performed to obtain the three interfaces:

1. ***Variation Interface***:
   (a) Find the distinct features that the framework provides.
   (b) Create a feature model based on the inherent dependencies among the framework features. Such dependencies include: *OR, XOR, mandatory, optional* and cross-tree constraints (*require* and *exclude*).
   (c) Create an impact model by analyzing the high-level goals of the framework.
2. ***Customization and Usage Interface***:
   (a) Determine which code elements (e.g., classes or methods) belong to which feature. Sometimes different features share some code elements.
   (b) Create a realization model which groups all API elements belonging to the same feature. Code elements shared by multiple features should be put into a shared parent realization model and extended by other realization models.
   (c) Determine and specify the usage protocols to use each class in a usage interface.
   (d) Identify the code elements which need to be added if a specific feature is selected. Add these code elements to the customization interface.

Once these interfaces constructed, the reuse process discussed in Section 2.1.2 can be applied to achieve the benefits of concernification. We summarize the benefits as followed:

- **Documents Features**: The variation interface of a concern (usually in the form of a feature model) structures the functionality of the framework at a high-level of abstraction, from the perspective of framework users. By reading the feature tree and their inter constraints, users can quickly grasp the framework functionality and the dependencies.
- **Tailor the APIs**: A framework provides comprehensive functionality thus usually exceeds the individual users' need. When facing the enormous options of APIs, users might have to spend some time digging out the most relevant APIs for their needs. With a concernified

framework, the user can make a feature selection in the variation interface and therefore access only a subset of the APIs corresponding to the user's needs. It greatly reduces the API complexity for users.

- **Guarantee Correct Reuse**: The customization interface forces the user to follow the rules or constraints to adapt the framework to the context of reuse. For example, to fulfill the *Observer Design Pattern*, the users need to provide a mapping from the *subjects* or *observers* to some elements in their application.

In Schöttle's work [23], he further proposed an automatic concernification algorithm to produce an initial concern interface. The automatic concernification algorithm works with a directed acyclic graph (DAG) based on the inputs of several code examples. Nodes represent the potential features and are processed in the algorithm based on the available information of the framework and examples. Each edge represents a multi-set of possible relationships between the features, where the possible relationships include: inheritance, containment, cross-reference, and structural-grouping.

2.1.4 TouchCORE

To put the concepts of CORE into practice, effective tooling is necessary. Such tools need to guide the users through the concern reuse process, and provide the support for interacting with several models involved (e.g., feature model and realization model). Furthermore, the tool has to take care of ensuring correct interactions with the concern interfaces (e.g., making correct feature selections, specifying correct customization mappings, and ensuring consistent use of the concern's structure and functionality exposed in the usage interface). The tool also takes care of composing the realization models based on feature selections (i.e., *weaving*). There exist tools for creating and editing feature models (such as *FeatureIDE* [1]) or different kind of realization models (such as *Papyrus* [2]). Some of them (such as *Clafer* [3]) even provide support for the combination of these two models.

In this thesis, we used the *TouchCORE* [4] as it directly supports CORE and the CORE reuse process [11]. TouchCORE was initially called TouchRAM [1] which exploits *Reusable Aspect Models* (RAM) to enable rapid application of reusable design concerns within design models of the software under construction. TouchRAM was then upgraded to support Concern-Oriented Reuse (CORE) (see Section 2.1.2 for more details on CORE) and consequently renamed TouchCORE.

[1]https://featureide.github.io/

[2]https://www.eclipse.org/papyrus/

[3]https://www.clafer.org/

[4]http://touchcore.cs.mcgill.ca/

The TouchCORE tool is built on top of Eclipse Modelling Framework (EMF) to define the abstract syntax for CORE and RAM models. These CORE and RAM models are structured and serialized in XMI (XML metamodel Interchange) and EMF enables Java code generation from them.

TouchCORE supports feature and impact models, as well as use case diagrams, class diagrams, sequence diagrams and state diagrams. It can be used by a concern designers to create a concern with its three interfaces. In the process of concernification, the complexity of the aspect-oriented model composition is hidden by TouchCORE as it transparently executes the aspect-oriented composition algorithm in the background. When a concern is reused by a concern users, TouchCORE streamlines the reuse process by guiding the concern user's interaction with the VCU interfaces of the concern.

2.2 Related Work

This thesis focuses on providing high-level, feature-oriented interface for DL libraries that allows newcomers to shrink the API of the framework to only those API elements that are related to the features of interest to the developer.

In general, the API of a library hides the implementation details and complexity of the underlying code from users and exposes only well-defined entry points to trigger the provided functionality. Unfortunately, such a simplification also creates a problem: with limited knowledge about the internals of the third-party library it is not always easy to use the API correctly. Even though the API documentation typically provides references and examples that help the users, these examples are often not comprehensive and don't cover all important cases.

This section reviews existing related work on API documentation augmentation, work on supporting the understanding and using of APIs, and work on raising library abstraction level.

2.2.1 Improving API Documentation

API documentation is often incomplete and can be augmented by information from crowd sources such as *Stack Overflow*, *GitHub*, websites, etc. Some of the existing approaches augment the API documentation by adding website reference to it. For example, *Subramanian et al.* [24] presented an iterative, deductive method called *Baker* for linking source code examples to API documentation, which fills the gap between traditional API documentation and example resources. The approach uses the answers from *Stack Overflow*, identifies the types and methods in the code snippets, and determines their fully qualified names. With this information, *Baker* can automatically augment the API documentation of the types and methods it detected by injecting the code sample into the API documentation's webpage. It can also add links to the official APIs into the *Stack Overflow* answers. This is realized by using a large database containing information about the code elements in popular APIs. The approach first generates an incomplete abstract syntax tree (AST) for the code snippet, and then uses information from the database to deduce facts about the AST nodes iteratively. The iteration stops when either all AST nodes are associated with a single fully qualified name, or it fails to improve the results for any AST node.

Similarly, *Uddin et al.* [27] proposed a framework to mine API usage scenarios from *Stack Overflow*, which presents the API documentation in a task-based view. Each task consists of a code example, the task description, and the reactions of developers (i.e., positive or negative opinions) towards the code example. The approach first links the code sample to the API mentioned in the *Stack Overflow* post with that code sample. Then, natural language processing is used to summarize the discussion around the code sample to generate its natural language description. At last, the approach analyzes the opinions of the people discussing the post towards the code example to offer information about code quality. The task related information is then presented to the users of the framework when they search for API usage scenarios.

Other approaches enrich the content of the API documentation, e.g., adding valuable sentences. *Treude et al.* [26] proposed an approach to automatically augment API documentation with "insight sentences" from Stack Overflow. Those sentences are related to a particular API and can provide insight not contained in the API documentation. To achieve this, *Treude et al.* developed SISE (Supervised Insight Sentence Extractor), a novel machine learning based approach that extracts related sentences from *Stack Overflow* and summarizes them as "insight sentences". With this tool, users can directly get the insight sentences right next to the API explanation in the documentation. SISE achieved a precision of 64% and a coverage of 70% on the development set. They also conducted a user study among eight software developers to compare SISE with other state-of-the-art text summarization approaches, where SISE resulted in having the highest number of sentences considered to provide information not included in the API documentation.

2.2.2 Support for Understanding and Using of APIs

Even though documentation can show the functionalities and usage of an API, according to *Gias et al.* [28], API documentation usually suffers from problems of bloated, scattered or tangled information. Therefore, a lot of studies focus on how to support developers to better acquire knowledge about APIs other than from API documentation.

For example, *Liu et al.* [15] proposed an approach for generating query-based class summaries, using an API knowledge graph (*API KG*). This method takes a natural language query Q and a class C from the existing library L as input. Based on that input, the API knowledge graph of the library L (i.e, *API KG(L)*) they constructed can extract up to S sentences describing C's functionality and up to M methods M_i in C which are most relevant to the query Q. To construct the API knowledge graph, *Liu et al.* first developed a web crawler to collect the API reference documentation. Then they developed a parser based on the structure of the collected API documentation to extract API entities and their properties and relations to build up the API knowledge graph. Using JDK and Android APIs as an illustration of their approach, the resulting API KG includes 137113 API entities and 305826 relation edges. To generate S sentences and M methods for a query Q, they defined a KG-based similarity metric to compute a relevance score between a user query and the API entities. Given a query Q, the KG-based similarity metric is a linear combination of a *Textual similarity* and a *Concept similarity* for each candidate API entity e: $Sim(Q, e) = w_1 \times Sim_{text}(Q, e) + w_2 \times Sim_{concept}(Q, e)$, where $w_1 + w_2 = 1$. *Textual similarity* considers the text context of the query and each API entity's documentation. *Concept similarity* measures the similarity between the semantic representations of their corresponding concepts, which can be learned from the API knowledge graph.

Likewise, *Ponzanelli et al.* utilized *Stack Overflow* to help developers comprehend and develop software [20]. They provided an *Eclipse* plugin named S_{EAHAWK} which can automatically formulate queries based on the current context in *Eclipse*. Users can interact with the queries results and import code samples simply by dragging and dropping. The plugin is composed of a data collection engine, a search engine, a recommendation engine, and a query engine using natural language processing technology. They built their database using a public data dump provided by*Stack Exchange* [5], which offers XML files to represent all user-contributed content for each website (e.g., *Stack Overflow*). Then the information was extracted from these XML files and converted into JSon documents for portability reasons. The search engine indexes these documents and makes them available for queries.

To help the users without prior knowledge and start learning APIs more quickly, *Yin et al.* designed an API learning service for inexperienced developers [30]. First, they proposed a method to link the API with its corresponding learning resources from *Stack Overflow*. Then, they constructed an API knowledge graph with the APIs and their related threads from *Stack Overflow*. At last, by mining how APIs are discussed in *Stack Overflow*, they proposed a learning entry recommendation method, where a learning entry is defined as a set of APIs commonly used by developers. The learning entry recommendation method is achieved by constructing a new knowledge graph which only contains the learning entry. Users can starts from any API in the learning entry and learn their related APIs following the links without having to input any query.

Recently, many studies focuses on recommending APIs according to natural language queries made by developers. For example, *Gu et al.* proposed *DeepAPI*, which is a deep learning method to generate a sequence of API usages based on a natural language query [8]. *DeepAPI* learns the sequence of the words in the query, and their associated API sequence. It is achieved by adapting a

[5]https://archive.org/details/stackexchange

model named RNN encoder-decoder. The model encodes the user query into a fixed-length vector, and generates the APIs sequence based on the vector. *Gu et al.* empirically evaluated the approach with more than 7 million annotated code snippets from *GitHub* and outperformed two state-of-the-art API learning approaches (i.e., Code Search with Pattern Mining [14] [29] and SWIM [21]).

Since it was published, more advanced techniques have been employed for code and natural language representation learning. For example, the work *CodeBERT* proposed by *Feng et al.* [6] demonstrates a better performance on several downstream tasks that require reasoning across natural language and programming languages according to Martin [16]. It is a pre-trained model built with *Transformer*-based neural architecture. As it learns general-purpose representation, it supports downstream NL-PL applications such as natural language code search, code documentation generation, etc. Results show that CodeBERT achieves state-of-the-art performance on both natural language code search and code documentation generation. Similarly, *Huang et al.* proposed *BIKER* [9](Bi-Information source based KnowledgE Recommendation), that uses the word embeddings technique to calculate the similarity score between two text descriptions: *Stack Overflow* posts and API documentation. After obtaining the ranked list of candidate APIs, *BIKER* summarizes supplementary information to help developers select the most relevant API. The evaluation with 413 API-related questions demonstrates the effectiveness of *BIKER* for both class and method level API recommendation.

2.2.3 Raising Framework Abstraction Level

The Concernification approach discussed in Section 2.1.3 is a way of raising the abstraction level of a framework. Benefiting from the raise of abstraction level, its reusable code artifact allows users to access the large amount of framework functionality. Additionally, it can also support the framework to be reused at the implementation level by understanding the framework functionality and their corresponding APIs. There are two works focusing on raising a framework abstraction level: *design fragments* and framework-specific modeling languages (FSML).

Design fragment is proposed by *Fairbanks et al.* [5] to help developers use a framework for achieving a specific goal. For example, a *Design fragment* could represent a design that uses the framework *DL4J* to train a multi-layer network using CSV data. Users need to provide some code elements and declare the relationship of those code elements they provided with the elements the framework provides, i.e., whether they must sub-class a class, implement an interface or override a method. These code elements and their related framework APIs are the *design fragments*. A *design fragment* consists of classes, interfaces, methods and fields. It can also include one or several behavioural specifications, e.g., the users need to create new instances or invoke methods.

The *design fragment* is defined using an XML schema definition (XSD). Furthermore, it can also include some free-form text for documentation reasons. The free-form text can add additional details which cannot be delivered by pure XSD, such as whether a method is a callback method and how often it is invoked. *Design fragments* defined in XSD can be read and processed by an Eclipse plugin provided by the authors. This Eclipse plugin can 1. display a catalog of *Design fragments*, 2. display a list of the *Design fragments* that associate to the source code, 3. display a set of problem markers that appear in the standard Eclipse problem view,

In general, design fragments specify a subset of the framework APIs and user-relevant code like concernification does. However, there are still some difference between design fragments and concernification. For example, the collection of code examples have no relationships (groupings, constraints, etc.) between them in design fragments. On the contrary, in concernification, the code examples are specifically associated with the framework's high-level features and the inter-relationships of the features are specified in the feature model. Also, the design fragments language definition only provides documentation text to distinguish callback methods and framework-provided methods. But in concernification, the framework-provided methods are defined in the usage interface while all callback methods are part of the customization interface.

Another approach for abstracting a library is proposed by Antkiewicz *et al.*, which introduces *framework-specific models* [2] as a way to help developers build a framework-based application. According to Antkiewicz *et al.*, "a *framework-specific model* is a representation of a framework-based application that is useful for answering questions related to the usage of the framework's API by that application". Those questions concentrate on the way that the application is using the framework. For example, in order to use the framework correctly, how should the application be using the framework, etc.

To express a framework-specific model, Antkiewicz *et al.* used *framework-specific modeling languages* (FSMLs). The FSML formalizes the set of abstractions that the APIs provides (referred as API concepts below), and the constraints imposed by the framework. FSML makes use of *cardinality-based feature model* [3] as modeling notation. The cardinality-based feature model enables features to have multiplicity. The features are closely related to the code by encoding the required code which need to be provided by the users, such as the class extending, instance instantiation. Furthermore, there exist mappings between features and code patterns (structural or behavioral) to define their correspondence. In this way, a framework-specific model using FSML represents a feature configuration to describe a framework-based application.

In contrast to the concernification described in Section 2.1.3, the feature model in FSML is more complicated and fine-grained as it contains variations, customization and usage steps to allow more flexibility in customization. In concernification, the variation interface encodes a high-level view of the framework and specifies user-perceivable functional features. The customization and usage are handled separately in their corresponding interfaces. Hence, the approach of FSML targets the experienced developers having knowledge in the framework, whereas the concernification approach is more suitable for the beginners of the framework. In the future, the possibility of combining these two approaches can be investigated to utilize both of their advantages. For example, we can provide a simple framework-specific model for the beginners. For the experienced developers, a more elaborate feature model can be provided to enable more flexible customization for the experienced developers in FSML.

2.2.4 Summary

In this Chapter, we first reviewed some background concepts. We started from Model-Driven Engineering (MDE) which advocates the use of different modeling formalisms and notations in different software phases. Among these modeling formalisms, we introduced feature models and class diagrams in detail. Then, we explained Concern-Oriented Reuse (CORE), an approach for packaging reusable software development artefacts while at the same time providing Separation of Concerns (SoC). At the end of the background section, we brought up the concept of *Concernification*, an approach to raise the abstraction level of a framework and modularize a framework's API according to user-relevant functional features.

In the related work section, we reviewed some existing related work on API documentation augmentation, work on supporting the understanding and using of APIs, and work on raising library abstraction level. Most of the work in supporting beginners using API utilizes machine learning technology and achieved relatively good results. To raise the abstraction level of a framework, to the best of our knowledge, there exists no other approach that addresses all the benefits of concernification.

To sum up, most existing approaches for API recommendation require the users to provide a research query. However, for an inexperienced developer with little prior knowledge about the library or framework, it could be challenging to write a good query. Furthermore, according to *Robillard and DeLine* [22], one of the obstacles faced by developers who are trying to learn new APIs is the lack of knowledge about the API's high-level design. Our work fills such a gap by using *Concernification* to present the overall architecture and functionalities of a framework with a feature model. Beginners can select the desired features and get the API recommendation based on their needs directly by interacting with the concern interfaces.

3
Concernification of a DL Library

This chapter presents the details of our approach by applying it to a concrete DL library: Deep Learning For Java (*DL4J*). Section 3.1 explains why we chose DL4J to showcase our approach. Section 3.2 presents the preparation process for the concernification of DL4L, as well as what information was consulted for building the concern-oriented model. It is followed by the demonstration of the feature model for DL4J in Section 3.3), its corresponding realization model in Section 3.4, and how we address feature interactions in Section 3.5. Section 3.6 illustrates how the final concern-oriented API can be used by beginners.

3.1 Deep Learning for Java

As claimed by a survey published in 2019 [17], Java and Scala are not as popular in the DL/ML research community as Python. Since Java is not a mainstream languages in the field of Machine Learning, the number of existing Java libraries for Machine Learning is limited compared to Python.

In this thesis, we chose *Deep Learning for Java* (DL4J) library to showcase our approach aimed at helping beginners with DL library APIs. DL4J is to our knowledge currently the main DL library used by the Java community. It is completely open-source, published using the Apache 2.0 licence and under open governance at the Eclipse Foundation. DL4J is still continuously updated by its contributors. Whenever a project requires the use of Java, but also needs to perform some DL task, many applications have chosen to use DL4J rather than paying the overhead of integrating Java and Python so that business functionality can be programmed in Java while DL-related functionality is handled in Python. In the end, *DL4J* has become the choice for many commercial industry-focused distributed DL platforms where Java is predominant. In github, the DL4J repository has gained over 12,000 stars. Besides its wide commercial usage, *DL4J* is also a popular library among open source projects, as demonstrated by the over 37,000 code examples using DL4J found in *Github*.

3.2 Concernification Preparation

DL4J comes with several documentation artefacts:

- Its source code in github[1];
- The API documentation[2];
- Runnable code examples[3];
- Its official tutorial website[4];
- A book [19] co-written by DL4J's main contributor Adam Gibson.

To concernify a framework, in-depth understanding of the framework is necessary. We first prepared ourselves with some basic understanding of *DL4J* library by reading its overview[5] from its official tutorial. Then we downloaded the official sample codes and ran 20 of them to gain an overall knowledge of the most commonly used APIs in specific use cases. In the procedure of concernifying the framework, we also referred to the official community forum[6] and some posts on the Internet.

DL4J has the following submodules:

- Datavec: A data transformation library converting raw input data to tensors suitable for neural networks;
- Nd4j: DL4J uses linear algebra and matrix manipulation as the basis of scientific calculation, which is realized by ND4J. It is similar to the functions that NumPy provides to Python and contains a mix of NumPy operations and TensorFlow/PyTorch operations in Java;
- Samediff: a TensorFlow/PyTorch like framework for the execution of complex graphs. This framework is lower level, but very flexible. It's also the base API for running ONNX (Open Neural Network Exchange, which is an open format built to represent machine learning models) and TensorFlow graphs;

[1]https://github.com/eclipse/deeplearning4j
[2]https://javadoc.io/doc/org.deeplearning4j/deeplearning4j-nn/1.0.0-M1/index.html
[3]https://github.com/eclipse/deeplearning4j-examples
[4]https://deeplearning4j.konduit.ai/
[5]https://deeplearning4j.konduit.ai/
[6]https://community.konduit.ai/

- Libnd4j: A lightweight, standalone C++ library that enables math code to run with good performance on different devices;
- Python4j: A python script execution framework easing the deployment of python scripts into production;
- Apache Spark Integration: An integration with the Apache Spark framework enabling the execution of deep learning pipelines on Spark.

Because of all the submodules, *DL4J* is a massive library, and the developers highly recommend using Maven to manage and run *DL4J*-related projects. Concernifying such a huge framework is out of the scope of a master thesis. Furthermore, because our work aims at helping beginner users of *DL4J*, we decided to only concernified a subset of *DL4J* library, a part that would most likely be used by the beginners. Since the developers recommend using Maven, the units of modularization of *DL4J* this thesis based on are `jar` files. Table 3.1 lists all the `jar` files used for the concernification of *DL4J* described in this thesis. The version of *DL4J* we used to build our model is 1.0.0-M1.1 unless the latest version for some specific *jar* files is 1.0.0-beta7 unless the latest version for some specific jar files is 1.0.0-beta7.

Neural Network Implementations	deeplearning4j-core-1.0.0-M1, deeplearning4j-nn-1.0.0-M1
Loading and Vectorizing Data	datavec-api-1.0.0-M1, datavec-data-audio-1.0.0-beta7, datavec-data-codec-1.0.0-beta7, datavec-data-image-1.0.0-M1, deeplearning4j-datasets-1.0.0-M1, datavec-data-NLP-1.0.0-beta7, deeplearning4j-datavec-iterators-1.0.0-M1
Computing Library for JVM	nd4j-api-1.0.0-M1, nd4j-common-1.0.0-M1, nd4j-native-1.0.0-M1, nd4j-native-api-1.0.0-M1, nd4j-parameter-server-1.0.0-M1

Table 3.1: Scope of the Concernification of DL4J

To simplify the process, we extracted the contents in these jar files, and then combined them into a single jar file called *DL4J_ CORE*.jar. For this jar file, we elaborated a feature model using the example codes and official tutorials. The feature model was iteratively refined until it reflected the *DL4J* library's functionalities from the user's perspective. To give the beginners an overview of the library functionalities properly, the feature model is constructed in a machine learning pipeline including the features of data preparation, network specification, and network operation.

Inside this *DL4J_ CORE*.jar, there are 3741 public classes and 92196 public functions (i.e., normal methods and constructors). It is still a huge library even after we narrowed down the scope by excluding some advanced functionalities. In Chapter 4, we used these two figures which reflects the original library size, and compared them with the number of classes and functions returned by our concernified library in order to showcase the concern's ability in reducing the API elements exposed to the users.

3.3 Feature Model

As presented in Section 2.1.3, the first step of concernification involves identifying the user-perceivable features and creating a feature model based on the inherent dependencies among the framework features. In this section, we present the feature model that we elaborated manually for *DL4J*. The feature model tree is organized into subtrees that follow the order of a typical machine learning pipeline: from loading the data, to data operations, to specifying the network and then running it. In the following, we will present each subtree in a separate subsection.

Figure 3.1 presents the complete feature model. As the legend shows, the filled circle means the corresponding feature is mandatory, while the hollow circle marks the feature is optional. Furthermore, a parent feature can group a set of features using an OR relationship (depicted with a filled arc), which means that at least one of the child features must be chosen. Finally, an XOR relationship among child features, depicted with a hollow arc, means that exactly one of the child features must be chosen.

Also, the feature model specifies some cross-tree constraints. These constraints stem from the logic of the machine learning pipeline, e.g., to evaluate a neural network, the network needs to be trained first, and only then the inference process can be executed. Other constraints are due to our simplification of the library, for example, we only provide data operations for formats of data that are likely to be used by beginners.

3.3.1 Load Data

Data loading and preparation is a pivotal phase in machine learning [31]. It ensures accuracy in the data, which somehow influences the model quality. In *DL4J*, the submodule named *Datavec* handles the Extract, Transform, Load (ETL) process and vectorization component in a machine learning pipeline [19]. In this section, we introduce the data loading procedure and its corresponding concernified features.

3.3 Feature Model

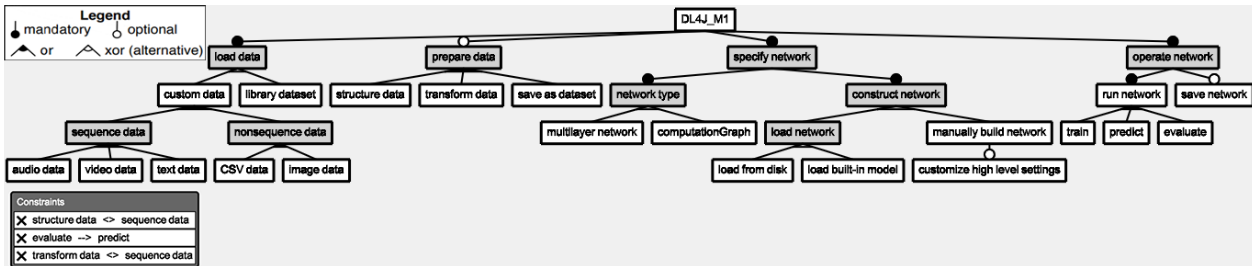


Figure 3.1: Feature Model for Deeplearning4J

In most cases, data for machine learning starts in a human-readable format. For example, there are formats like text data, image data, and audio data which are regularly processed by human eyes, ears, and brains in daily life. These formats of data cannot be directly fed into a Neural Network but must first be transformed into a vector representation. In *DL4J*, the process of loading data not only reads the raw data from disk, but also transforms the raw data into vectors. For simplification, we only provides 5 frequently used data types: text, audio, video, CSV, and image types. Since in some situations the developer might have data in multiple formats, the relationships between the data types in the feature model is specified as *OR*.

The users who want to test the machine learning algorithm on their own datasets, they need to convert their original raw data into vectors. Usually the raw data is in a single format, hence the *Datavec* module offers different readers to load each specific format of data. Since in this thesis we focus on simplifying the *DL4J* library API for beginners, we only take the most common data formats, i.e., CSV data, text data, image data, audio data, and video data, into account. We furthermore classified the data according to whether they are sequential or not.

For the beginners, toy datasets are pivotal to get started with learning the basic concepts of machine learning. D4LJ comes with a library of built-in datasets, including but not limited to:

- **MNIST database** [13] (Modified National Institute of Standards and Technology database) is a handwritten digits dataset, containing a training set of 60,000 examples, and a test set of 10,000 examples. It is one of the most commonly used databases for benchmarking machine learning algorithms.

- **Iris dataset** [7] consists of 50 samples from each of three species of Iris flowers. Four features were measured for each sample: the length and the width of the sepals and petals. This dataset became a typical test case for many statistical classification techniques in machine learning.

- **TinyImageNet** (subset of ImageNet [4]) contains 100000 images of 200 kinds of objects

(500 for each object) downsized to 64×64 colored images. Each kind of object has 500 training images, 50 validation images and 50 test images. Since it is a subset of *ImageNet*, researchers can use *TinyImageNet* to get an understanding of their models' ability more quickly.

- **CIFAR-10** [12] is a collection of images that are commonly used to train machine learning and computer vision algorithms, consisting of 60000 color images with the size of 32x32 categorized into 10 kinds of objects, with 6000 images per object.

Figure 3.2 shown below presents a closer look at the subtree rooted at the feature *load data*.

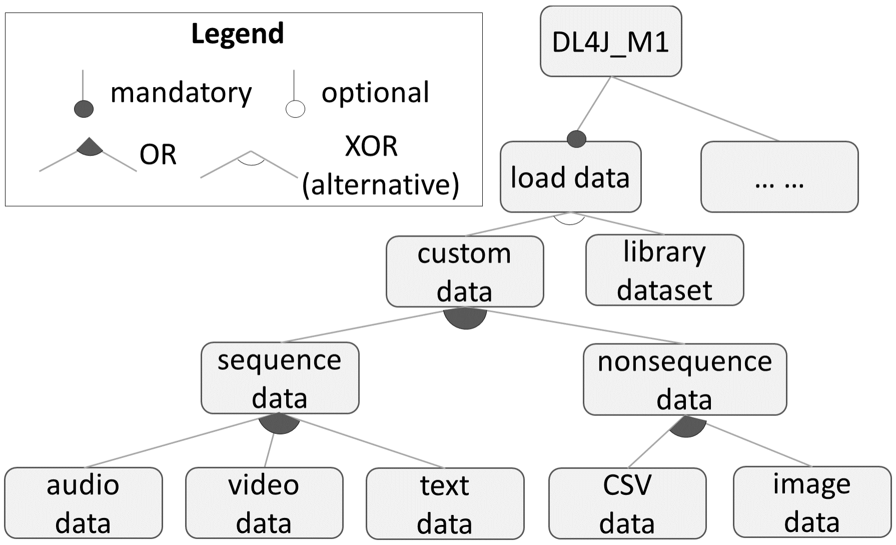


Figure 3.2: A subtree of the DL4J Feature Model related to Loading Data

As illustrated in Figure 3.2, selecting at least one of the subfeatures of *load data* is mandatory since Neural Networks always require data for training or prediction. Users can either choose to use their own raw data or use the library's built-in datasets as shown by the "XOR" children of *load data*.

3.3.2 Prepare Data

In the field of machine learning, preparing data means loading it and often processing it to put it into the right format and/or adjust the value range. In the previous section we have already discussed how to load the data. After loading, the raw input data is automatically transformed into a Neural Network readable format called *vector*, which can already be fed into a Neural

Network for training. Because of this, the *operate data* subtree is not mandatory in our feature model. However, in practice, some data modification operations are usually performed on the data to increase the data quality, since raw data are often error-prone.

In our feature model, to make it easier for beginners to understand, we classified the optional data operations into 3 categories: structuring data, transforming data, and saving data.

- **Structuring data** means changing the dimension of vectors. For CSV data, for example, we can add or delete a column, which results in modifying the shape of the data. If we consider image data as a 2-dimensional array, operations like resizing, flipping, rotating or cropping are changing the dimension of it, which also falls into the category of structuring data.

- **Transforming data** means changing the value range or value type of data. Take CSV data as an example. It is possible to scale the values of a numeric column into a certain range using the formula $x^{'} = (x - x_{min})/(x_{max} - x_{min})$. We can also convert a categorical value into an integer value. In image processing, normalization is a process that changes the range of pixel intensity values. These kinds of operations modify the value range of the data.

- **Saving data** refers to saving the processed data onto the local hard disk in case the data are needed in the future in processed form. In this way, users only need to load the processed data directly and use them without having to process them again.

Even though audio data, video data and text data can be structured and transformed, considering the fact that machine learning beginners usually do not start from that kind of data, we built our corresponding realization model only for CSV and image data. To express this decision in our feature model, we add a constraint that structuring or transforming data requires non-sequence data loaded.

As seen in Figure 3.3, the "operate input data" feature is optional, shown by the hollow circle on the relationship. This means that users can choose to not select it. If they decide to choose it, then they should choose one or several among the features "structure data", "transform data" and "save data".

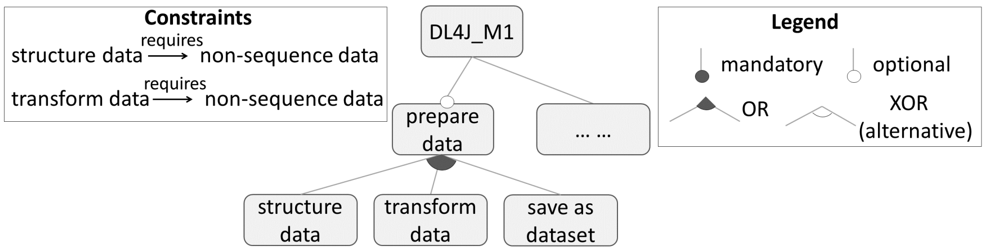


Figure 3.3: Preparing Data Part in the DL4J Feature Model

3.3.3 Specify Network

After the preparation of the data comes the phase of specifying the Neural Network. In this phase, users need to consider two things ahead:

- The type of the Neural Network; *DL4J* has two types of networks comprised of multiple layers: the *MultiLayerNetwork*, which is essentially a stack of neural network layers (with a single input layer and single output layer), and the *ComputationGraph*, which allows for greater freedom in network architectures.
- The way the Network is being built; *DL4J* supports loading a saved model from the disk, loading a built-in model, or manually constructing the network layer by layer.

MultiLayerNetwork is the kind of network most frequently used by beginners who do not need a complex and branched network graph. It is a simple and sequential model, as opposed to the *ComputationGraph*, which allows multiple layers of inputs and outputs, and a directed acyclic graph connection structure between the layers.

After the selection of the network type, users can construct the network either by loading it or by manually building it. *MultiLayerNetwork* and *ComputationGraph* both have save and load methods. Therefore, a user can load the network from disk and recover it for further training or move on to using the network for prediction or evaluation.

Another loading choice is loading a built-in network provided by the *DL4J* framework. *DL4J* comes with a library of existing models called *model zoo* [7] that can be accessed and instantiated directly. The model zoo includes pre-trained weights for different datasets that are downloaded automatically. The model zoo comes with well-known image recognition configurations in the deep learning community. The zoo also includes an LSTM for text generation, and a simple CNN for general image recognition. There are 16 types of networks included in the model zoo. In the feature model, they are classified into *MultiLayerNetwork* and *ComputationGraph*, which means to use them, the users need to know what type they belong to.

To manually build a Neural Network, its architecture is the first and foremost consideration. The Neural Network architecture is made of individual units called neurons that mimic the biological behavior of the brain. Each layer in a neural network configuration represents a group of hidden units. When layers are stacked together, they represent a deep neural network. The creativity of the famous networks (e.g., AlexNet, CNN, VGG...) comes from their architectures. All layers available in *DL4J* can be used either in a *MultiLayerNetwork* or *ComputationGraph*. When

[7] https://deeplearning4j.konduit.ai/deeplearning4j/reference/model-zoo

configuring a neural network, the user provides the layer configuration as input and the network instantiates the specified layer automatically. What is special about the *ComputationGraph* network is that since it offers greater freedom in the network architecture, some branches of layers can be joined together using vertices.

To understand a neural network, apart from its architecture, its parameters and hyper-parameters are also critical.

- **Parameters** are the coefficients of the model, and they are chosen by the model itself and updated during the learning process.

- **Hyper-Parameters** (e.g., activation function, momentum, minibatch size, epochs, learning rate...) are not updated during the training process. Therefore, initialization is needed to enable the best network performance.

In our feature model, we call these hyper-parameters *high-level settings*.

The feature model subtree related to specifying the Neural Network was built to reflect the discussion above. As illustrated in Figure 3.5, the feature "specify network" and the two sub-features "network type" and "construct network" are mandatory. Since *DL4J* provides default settings for the hyper-parameters, the feature "customize high-level settings" is optional.

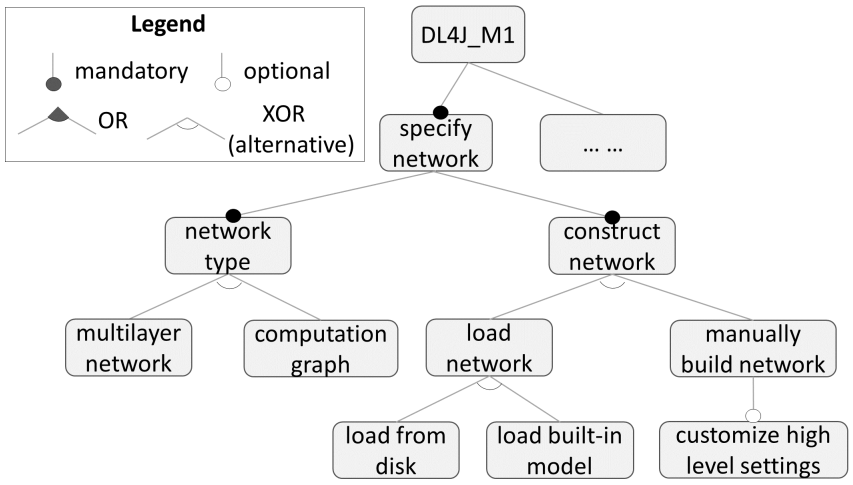


Figure 3.4: Specifying Network Part in DL4J Feature Model

3.3.4 Operate Network

The most important phase in the machine learning pipeline is to operate the network, which involves training, predicting and evaluating. Training refers to the process where a machine learning network learns the patterns contained in sufficient training data for learning a specific task. Evaluation of a neural network involves calculating a performance measure that indicates how well the network is trained. The measurement is obtained by comparison of the network output with known labels on some test data. If the performance measure meets the expectation, the trained network can be applied for prediction on unknown data. Considering the main purpose of using *DL4J* is to run the network for the three processes discussed above, we made the feature "run network" mandatory.

Apart from running the network as discussed above, *DL4J* also provides an API to store a network on disk, i.e., saving the network architecture and its parameters. This works for trained, half-trained or untrained networks. The differences between the networks with the same architecture are their parameter values. Untrained networks have the default parameters when they were initialized. With the iteration of the epochs and learning process, the parameters evolve according to the algorithm. If the users want to use current model for further training or prediction, they can save it on disk and load it when necessary.

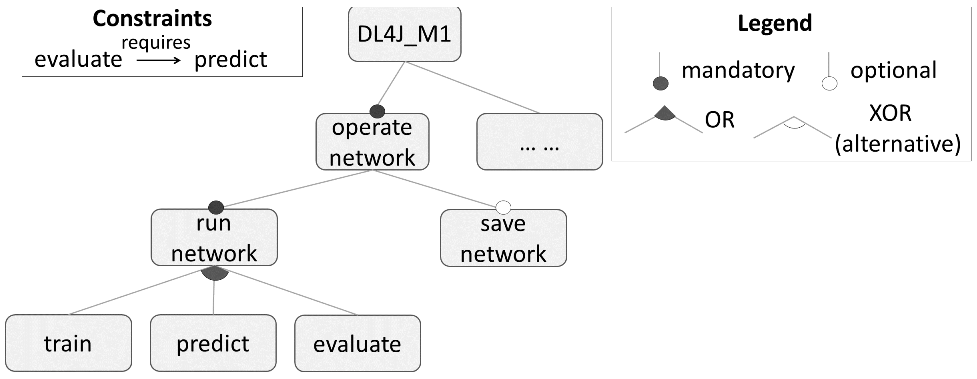


Figure 3.5: Operating Network Part in DL4J Feature Model

3.4 Realization Models

In this section, we are going to talk about how to modularize the API elements (classes, methods, constructors) of DL4J into realization models that are attached to their corresponding features. In this way, a desired subset of the DL4J API elements based on the feature selection can be generated. We will discuss the essential API elements for each features first by presenting some code snippets from the official DL4J examples in its GitHub repository. After that, we will show the realization models containing these API elements.

3.4.1 Data Loading

The two essential classes in the process of data operation are *RecordReader* and *DataSetIterator*.

- The *RecordReader* is a class in the *DataVec* module that helps convert byte-oriented input into data that is represented in form of records, i.e., a collection of elements that are fixed in number and indexed with a unique ID. Converting data to records is the process of vectorization. The record itself is a vector, each element of which is called a feature[8]

- The *DataSetIterator* is a *DL4J* class that traverses the elements of a list. An iterator passes through the data list, accesses each item sequentially, keeps track of how far it has progressed by pointing to its current element, and modifies itself to point to the next element with each new step in the traversal.

Datavec supports the formats of tabular (comma-separated values [CSV] files, etc.), image, and time-series datasets, both for single machine and distributed (Apache Spark) applications. Users can use the corresponding *RecordReader* and *DataSetIterator* to load their datasets. Here is a code snippet illustrating the APIs of *RecordReader* and *DataSetIterator*.

```
// Instantiating RecordReader. Specify height, width and channels of images.
// Note that for grayscale output, channels = 1, whereas for RGB images,
   channels = 3
RecordReader recordReader = new ImageRecordReader(28, 28, 3);

// Point to data path.
recordReader.initialize(new FileSplit(new File(labeledPath)));

// DataVec to DL4J
DataSetIterator iter = new RecordReaderDataSetIterator(recordReader, 784,
   labels.size());
```

Listing 3.1: Example of Loading Image Data

Figure 3.6 is the corresponding realization model containing the API elements in the above code snippet.

[8]The term *feature* used here is from the world of Machine Learning, and should not be confused with the *features* found in a feature model as used in Software Product Lines.

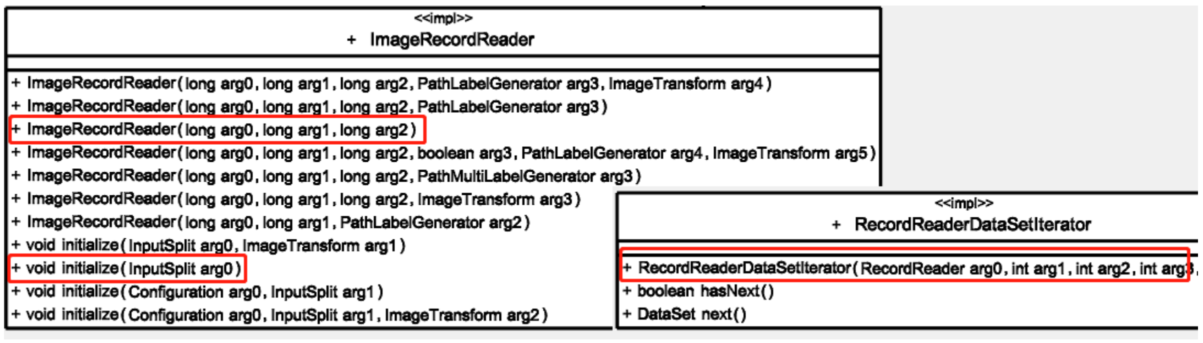

<<impl>>
+ ImageRecordReader
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3, ImageTransform arg4)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2)
+ ImageRecordReader(long arg0, long arg1, long arg2, boolean arg3, PathLabelGenerator arg4, ImageTransform arg5)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathMultiLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2, ImageTransform arg3)
+ ImageRecordReader(long arg0, long arg1, PathLabelGenerator arg2)
+ void initialize(InputSplit arg0, ImageTransform arg1)
+ void initialize(InputSplit arg0)
+ void initialize(Configuration arg0, InputSplit arg1)
+ void initialize(Configuration arg0, InputSplit arg1, ImageTransform arg2)
<<impl>>
+ RecordReaderDataSetIterator
+ RecordReaderDataSetIterator(RecordReader arg0, int arg1, int arg2, int arg3,
+ boolean hasNext()
+ DataSet next()

Figure 3.6: Realization Model of Loading Image Data (Partial)

For the built-in datasets, DL4J provides APIs that directly access them. For example, *MnistDataSetIterator* is an API that provides simple access to the MNist data set, even automatically downloading data in the background. It extends the *DataSetIterator* class.

Here is a code snippet using it:

```
//Get the DataSetIterators:
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize, true,
    rngSeed)
```

Listing 3.2: Example of Using Library Dataset

Also, in some cases, users might have more than one data format they want to work with. Hence, there are two classes designed for this situation: *ConcatenatingRecordReader* and *ComposableRecordReader*. These classes can integrate different readers so that users can use them to load multiple types of data.

Figure 3.7 illustrates how these two classes should be used depending on the source data format. In the illustrated example there are three types of data: Image data, CSV data, and Text data. Their corresponding readers are *ImageRecordReader*, *CSVRecordReader*, and *LineRecordReader*.

ConcatenatingRecordReader can combine these three readers into one single reader. That combined reader reads the data sequentially in the order of the contained readers. The numbers of data records of each type does not necessarily have to be the same. In the example illustrated in the figure, the first reader reads 3 images, the second reader reads 3 CSV records, while the third reader only reads 2 text records. As a result, the *ConcatenatingRecordReader* will read 8 records.

The *ComposableRecordReader* operates differently. It iterates over each readers reading only one record each, and then combining the read records into an individual record. Thus, the number of records of each reader should be the same. In the example, the *ComposableRecordReader* will have the same number of records as its composed readers, which is 3.

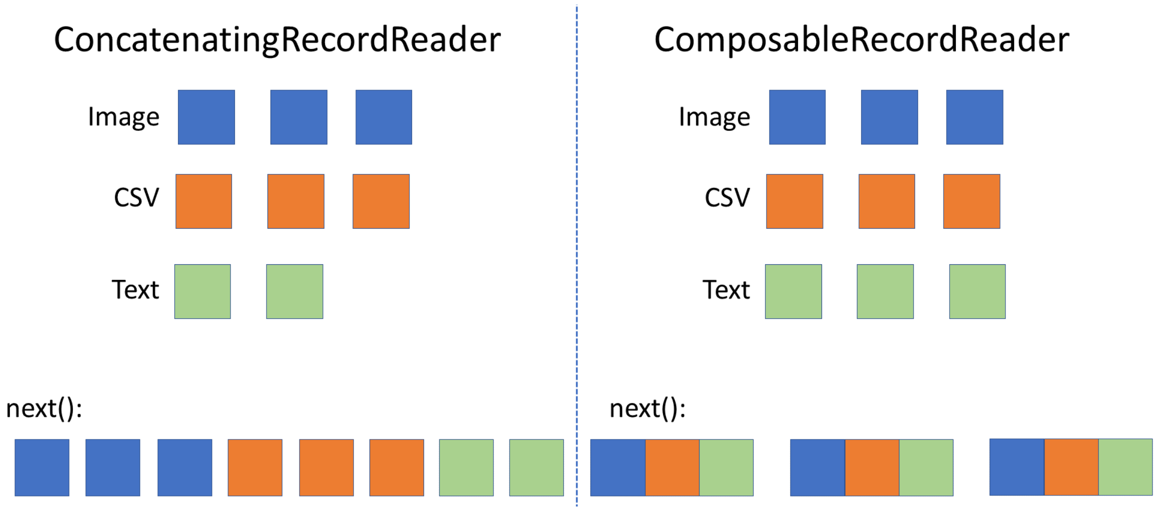


Figure 3.7: An example illustrating the difference between ComposableRecordReader and ConcatenatingRecordReader

The API provided by the *ConcatenatingRecordReader* and *ComposableRecordReader* is identical. Since the constructors of both classes accept a variable number of arguments (zero or more), in practice, users only need to pass all the RecordReaders corresponding to the data formats they want to process to the constructor when instantiating one of these classes.

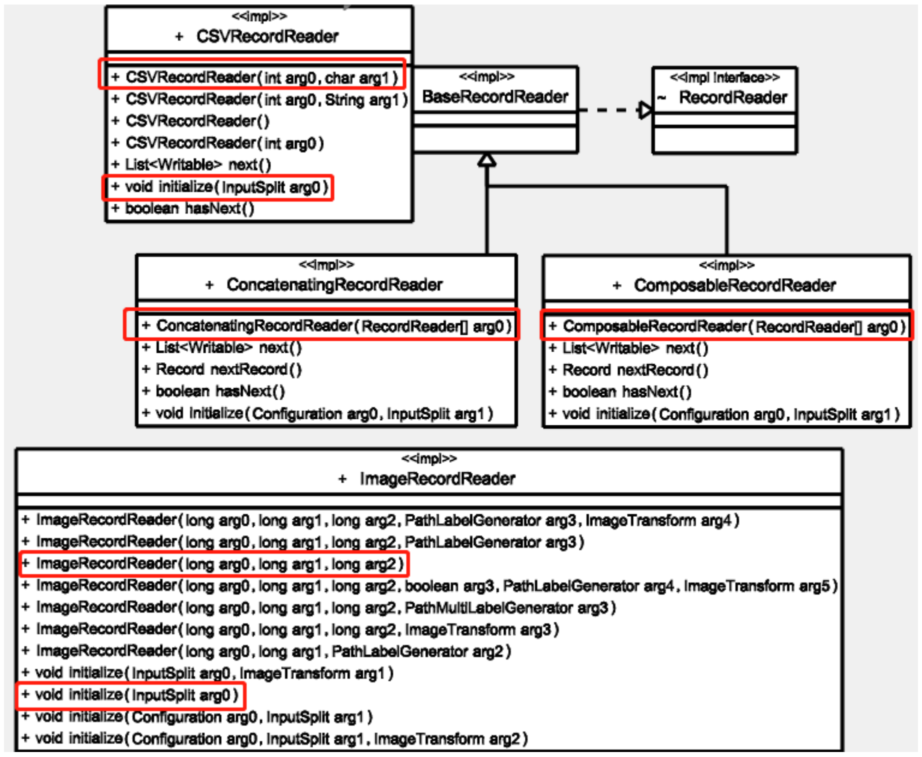
```
// First RecordReader for CSV data
CSVRecordReader rr = new CSVRecordReader(0, ',');
rr.initialize(new FileSplit(new ClassPathResource("iris.csv").getFile()));

// Second RecordReader for Image data
RecordReader rr2 = new ImageRecordReader(28, 28, 3);
rr2.initialize(new FileSplit(new File(labeledPath)));

// ConcatenatingRecordReader
RecordReader rrConcatenating = new ConcatenatingRecordReader(rr, rr2);
// ComposableRecordReader
RecordReader rrComposable = new ComposableRecordReader(rr, rr2);
```

Listing 3.3: Example of Loading Multiple format of Data

Figure 3.8 is the corresponding realization model containing the API elements in the above code snippet.



<<impl>>
+ CSVRecordReader
+ CSVRecordReader(int arg0, char arg1)
+ CSVRecordReader(int arg0, String arg1)
+ CSVRecordReader()
+ CSVRecordReader(int arg0)
+ List<Writable> next()
+ void initialize(InputSplit arg0)
+ boolean hasNext()
<<impl>>
BaseRecordReader
<<impl Interface>>
~ RecordReader
<<impl>>
+ ConcatenatingRecordReader
+ ConcatenatingRecordReader(RecordReader[] arg0)
+ List<Writable> next()
+ Record nextRecord()
+ boolean hasNext()
+ void initialize(Configuration arg0, InputSplit arg1)
<<impl>>
+ ComposableRecordReader
+ ComposableRecordReader(RecordReader[] arg0)
+ List<Writable> next()
+ Record nextRecord()
+ boolean hasNext()
+ void initialize(Configuration arg0, InputSplit arg1)
<<impl>>
+ ImageRecordReader
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3, ImageTransform arg4)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2)
+ ImageRecordReader(long arg0, long arg1, long arg2, boolean arg3, PathLabelGenerator arg4, ImageTransform arg5)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathMultiLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2, ImageTransform arg3)
+ ImageRecordReader(long arg0, long arg1, PathLabelGenerator arg2)
+ void initialize(InputSplit arg0, ImageTransform arg1)
+ void initialize(InputSplit arg0)
+ void initialize(Configuration arg0, InputSplit arg1)
+ void initialize(Configuration arg0, InputSplit arg1, ImageTransform arg2)

Figure 3.8: Realization Model of Loading Multiple format of Data (Partial)

3.4.2 Data Operation

As discussed in section3.3.1, we only included CSV and image data operations in the API we are generating, considering the fact that this is what beginners use most of the time.

CSV Data Operations

As a reminder, in our feature model the data operations are modularized into operations for structuring the data, and for transforming the data.

Structuring CSV Data To structure the CSV data, there are two classes involved: *Schema* and *TransformProcess*. Schemas are used to describe the layout of tabular data. The users need to use *Schema* first to define the layout of their current data.

```
Schema inputDataSchema = new Schema.Builder()
    .addColumnsString("DateTimeString", "CustomerID", "MerchantID")
    .addColumnInteger("NumItemsInTransaction")
```

```
   .addColumnCategorical("MerchantCountryCode",
      Arrays.asList("USA","CAN","FR","MX"))
   .addColumnDouble("TransactionAmountUSD",0.0,null,false,false)
   .addColumnCategorical("FraudLabel", Arrays.asList("Fraud","Legit"))
   .build();
```

Listing 3.4: Example of Defining CSV Data Schema

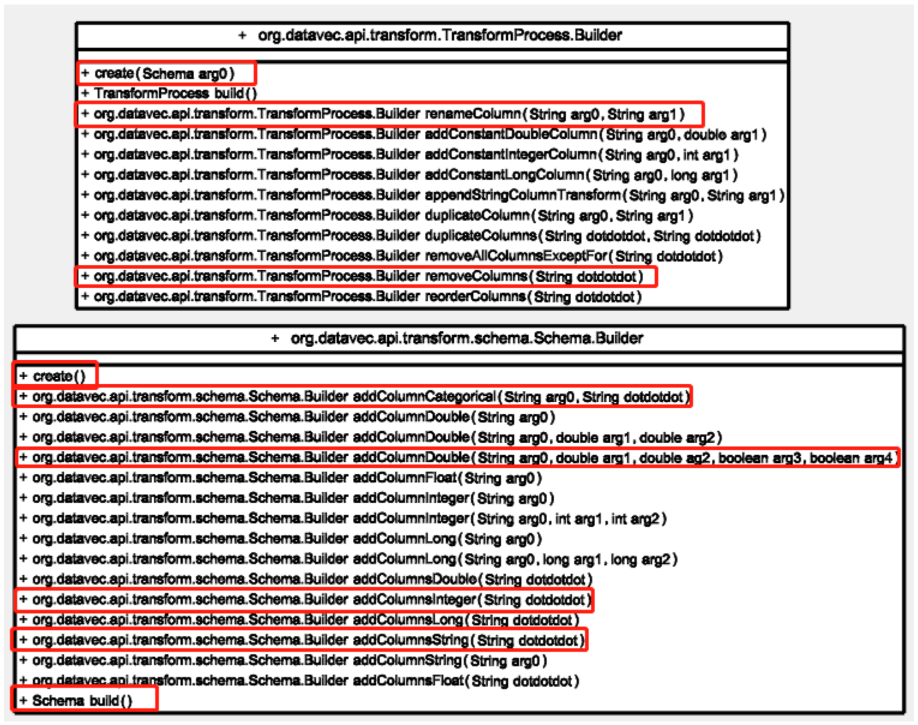
In this code example, *Schema* describes the layout of the data by adding columns. These columns are also specified with the data type (e.g, String, Integer, Double or Categorical). Note that the order of the columns should be exactly the same as the original data so that the Schema won't have divergence with the original data.

After defining the data layout, imagine the situation when the users want to remove certain unnecessary columns or rename columns, this is where *TransformProcess* engages.

```
TransformProcess tp = new TransformProcess.Builder(inputDataSchema)
   .removeColumns("CustomerID","MerchantID")
   .renameColumn("DateTimeString", "DateTime")
   .build();
```

Listing 3.5: Example of Structuring CSV Data

Figure 3.9 is the corresponding realization model containing the API elements in the above code snippet.



```
+ org.datavec.api.transform.TransformProcess.Builder

+ create(Schema arg0)
+ TransformProcess build()
+ org.datavec.api.transform.TransformProcess.Builder renameColumn(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder addConstantDoubleColumn(String arg0, double arg1)
+ org.datavec.api.transform.TransformProcess.Builder addConstantIntegerColumn(String arg0, int arg1)
+ org.datavec.api.transform.TransformProcess.Builder addConstantLongColumn(String arg0, long arg1)
+ org.datavec.api.transform.TransformProcess.Builder appendStringColumnTransform(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder duplicateColumn(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder duplicateColumns(String dotdotdot, String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder removeAllColumnsExceptFor(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder removeColumns(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder reorderColumns(String dotdotdot)
```

```
+ org.datavec.api.transform.schema.Schema.Builder

+ create()
+ org.datavec.api.transform.schema.Schema.Builder addColumnCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0, double arg1, double arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0, double arg1, double ag2, boolean arg3, boolean arg4)
+ org.datavec.api.transform.schema.Schema.Builder addColumnFloat(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger(String arg0, int arg1, int arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong(String arg0, long arg1, long arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsDouble(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsInteger(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsLong(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsString(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnString(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsFloat(String dotdotdot)
+ Schema build()
```

Figure 3.9: Realization Model of Structuring CSV Data (Partial)

Transforming CSV Data To transform the CSV data, *TransformProcess* is also used. The class provides some APIs that can perform filtering or value replacing.

```
TransformProcess tp = new TransformProcess.Builder(inputDataSchema)
   .conditionalReplaceValueTransform(
         "TransactionAmountUSD", //Column to operate on
         new DoubleWritable(0.0), //New value to use when the condition is
             satisfied
         new
             DoubleColumnCondition("TransactionAmountUSD",ConditionOp.LessThan,
             0.0)) //Condition: amount < 0.0
   .stringToTimeTransform("DateTimeString","YYYY-MM-DD HH:mm:ss.SSS",
         DateTimeZone.UTC)
   .build();
```

Listing 3.6: Example of Replacing Value on CSV Data

Besides, neural networks work best when the data used for training is normalized, constrained to a range between -1 and 1. There are several classes for normalization strategies. But here we only cover the class *NormalizerStandardize*, which normalizes feature values (and optionally label values) to have 0 mean and a standard deviation of 1.

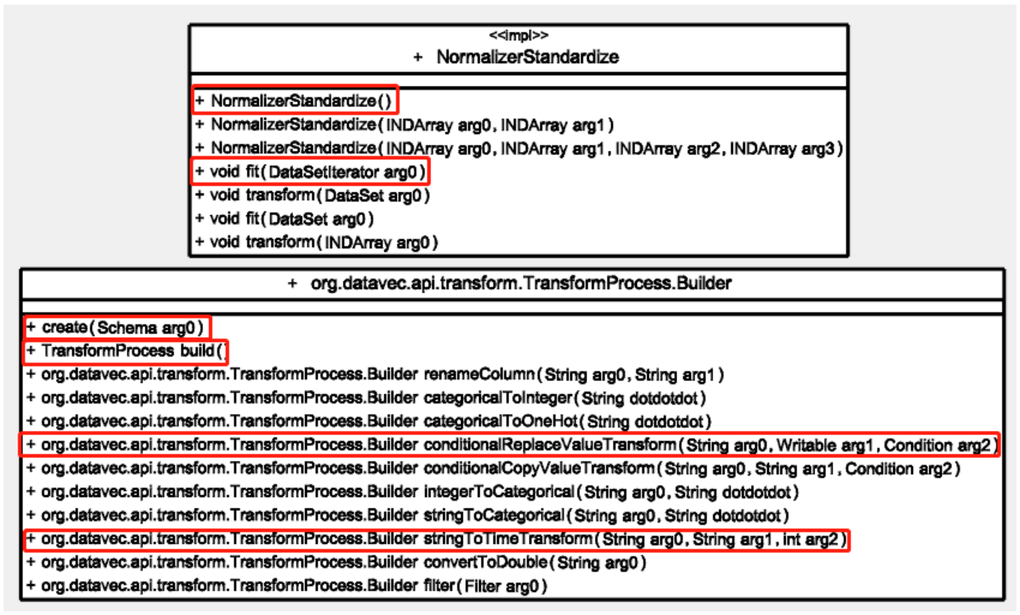
```
DataSetIterator trainData = new MnistDataSetIterator
    (batchSize, true, rngSeed);

// Normalize the training data
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData); // Collect training data statistics
trainData.reset();

// Use previously collected statistics to normalize. Each DataSet
// returned by 'trainData' iterator will be normalized
trainData.setPreProcessor(normalizer);
```

Listing 3.7: Example of Normalizing CSV Data

Figure 3.10 is the corresponding realization model containing the API elements in the above code snippet.



<<impl>>
+ NormalizerStandardize
+ NormalizerStandardize()
+ NormalizerStandardize(INDArray arg0, INDArray arg1)
+ NormalizerStandardize(INDArray arg0, INDArray arg1, INDArray arg2, INDArray arg3)
+ void fit(DataSetIterator arg0)
+ void transform(DataSet arg0)
+ void fit(DataSet arg0)
+ void transform(INDArray arg0)
+ org.datavec.api.transform.TransformProcess.Builder
+ create(Schema arg0)
+ TransformProcess build()
+ org.datavec.api.transform.TransformProcess.Builder renameColumn(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder categoricalToInteger(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder categoricalToOneHot(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder conditionalReplaceValueTransform(String arg0, Writable arg1, Condition arg2)
+ org.datavec.api.transform.TransformProcess.Builder conditionalCopyValueTransform(String arg0, String arg1, Condition arg2)
+ org.datavec.api.transform.TransformProcess.Builder integerToCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder stringToCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder stringToTimeTransform(String arg0, String arg1, int arg2)
+ org.datavec.api.transform.TransformProcess.Builder convertToDouble(String arg0)
+ org.datavec.api.transform.TransformProcess.Builder filter(Filter arg0)

Figure 3.10: Realization Model of Transforming CSV Data (Partial)

Image Data Operations

Structuring Image Data In our model, structuring image data means changing the size of it, like resizing or cropping. Users can apply *ResizeImageTransform* and *CropImageTransform* to perform their tasks.

```
ResizeImageTransform rit = new ResizeImageTransform(28, 28);
ImageRecordReader reader = new ImageRecordReader(56, 56, 3, labelGenerator,
    rit);

CropImageTransform cit = new
    CropImageTransform(cropTop,cropLeft,cropBottom,cropRight);
ImageRecordReader reader2 = new ImageRecordReader(56, 56, 3, labelGenerator,
    cit);
```

Listing 3.8: Example of Structuring Image Data

Figure 3.11 is the corresponding realization model containing the API elements in the above code snippet.

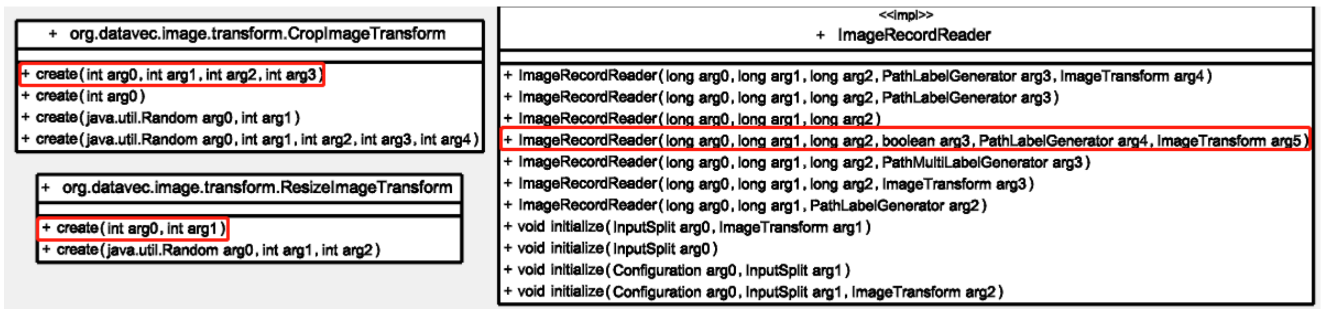

+ org.datavec.image.transform.CropImageTransform
+ create(int arg0, int arg1, int arg2, int arg3)
+ create(int arg0)
+ create(java.util.Random arg0, int arg1)
+ create(java.util.Random arg0, int arg1, int arg2, int arg3, int arg4)
+ org.datavec.image.transform.ResizeImageTransform
+ create(int arg0, int arg1)
+ create(java.util.Random arg0, int arg1, int arg2)
<<impl>>
+ ImageRecordReader
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3, ImageTransform arg4)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2)
+ ImageRecordReader(long arg0, long arg1, long arg2, boolean arg3, PathLabelGenerator arg4, ImageTransform arg5)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathMultiLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2, ImageTransform arg3)
+ ImageRecordReader(long arg0, long arg1, PathLabelGenerator arg2)
+ void initialize(InputSplit arg0, ImageTransform arg1)
+ void initialize(InputSplit arg0)
+ void initialize(Configuration arg0, InputSplit arg1)
+ void initialize(Configuration arg0, InputSplit arg1, ImageTransform arg2)

Figure 3.11: Realization Model of Structuring Image Data (Partial)

Transforming Image Data Transforming image data means changing the pixel value of the image. There are several classes in charge of this. *EqualizeHistTransform* flattens the intensity distribution curve, which can be used to improve the contrast of the image. *ImagePreProcessingScaler* can normalize the image pixel color values into a specified range.

Saving Data Operations To save the data, whether they are preprocessed or not, CSV or Image data, the *DataSet* class is responsible for it. *DataSet* is the return type of the *DataSetIterator.next()*. Thus, no matter whether the data is preprocessed or not, as long as the users obtains a *DataSet* object, it can be directly fed into the Neural Network or saved to disk.

```
// Example of CSV data
```

```
CSVRecordReader rr = new CSVRecordReader(0, ',');
rr.initialize(new FileSplit(new ClassPathResource("iris.csv").getFile()));
DataSetIterator iter = new RecordReaderDataSetIterator(rr, 784,
    labels.size());
int counter = 1;

while (iter.hasNext()) {
   String path = FilenameUtils.concat(dir, "dataset-" + (counter + ".bin");
   iterator.next().save(new File(path));
   counter += 1;
}

// Example of Image data
DataSetIterator iterator = new MnistDataSetIterator(batchSize, true, rngSeed);
int counter = 1;

while (iterator.hasNext()) {
   String path = FilenameUtils.concat(dir, "dataset-" + (counter + ".bin");
   iterator.next().save(new File(path));
   counter += 1;
}
```

Listing 3.9: Example of Saving Data

Figure 3.12 is the corresponding realization model containing the API elements in the above code snippet.

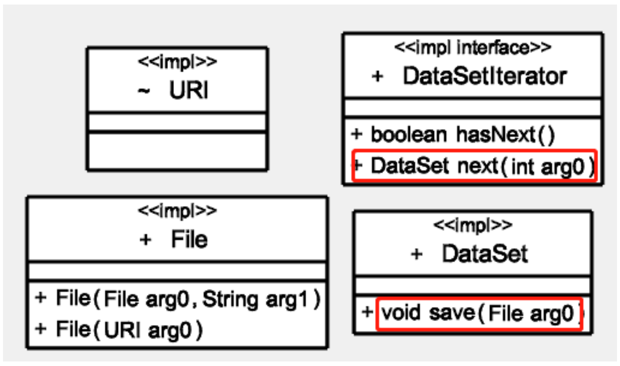


Figure 3.12: Realization Model of Saving Data (Partial)

3.4.3 Specify Network

As discussed in the section 3.3.3, *MultiLayerNetwork* and *ComputationGraph* are the most frequently used classes. To build them, their configurations (hyper-parameters and layers) need to be specified first. And then pass these configurations to their objects during instantiation. At last, using the API *MultiLayerNetwork.init()* or *ComputationGraph.init()* to finalize the building process.

```
// Configuration of MultiLayerNetwork
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
   .updater(new Sgd(0.01))
   .list()
   .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
         .weightInit(WeightInit.XAVIER)
         .activation("relu")
         .build())
   .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
         .weightInit(WeightInit.XAVIER)
         .activation("softmax").weightInit(WeightInit.XAVIER)
         .nIn(numHiddenNodes).nOut(numOutputs).build())
   .pretrain(false).backprop(true).build();

// Pass the configuration to a MultiLayerNetwork
MultiLayerNetwork multiLayer = new MultiLayerNetwork(conf);
multiLayer.init()
```

Listing 3.10: Example of Configuring MulitLayer Network

In the code snippet of the *MultiLayerNetwork*, after setting the hyper-parameter (i.e., updater), it uses the API *NeuralNetConfiguration.Builder().list()* to collect the stack of layers. Since the layers of a multilayer network are sequential, it just needs to add the layers orderly.

However, in a computation graph, the API *NeuralNetConfiguration.Builder().graphBuilder()* is used to build the graph. For example,

- *ComputationGraphConfiguration.GraphBuilder.addInputs(String... inputNames)* specifies the inputs to the network, and their associated labels.
- *ComputationGraphConfiguration.GraphBuilder.addLayer(String layerName, Layer layer, String... layerInputs)* add a layer with the specified name and specified inputs.
- *ComputationGraphConfiguration.GraphBuilder addVertex(String vertexName, GraphVertex vertex, String... vertexInputs)* adds a *GraphVertex* to the network configuration.

- *ComputationGraphConfiguration.GraphBuilder setOutputs(String... outputNames)* sets the network output labels.

```
// Configuration of ComputationGraph
ComputationGraphConfiguration conf = new NeuralNetConfiguration.Builder()
      .updater(new Sgd(0.01))
   .graphBuilder()
   .addInputs("input1", "input2")
   .addLayer("L1", new DenseLayer.Builder().nIn(3).nOut(4).build(), "input1")
   .addLayer("L2", new DenseLayer.Builder().nIn(3).nOut(4).build(), "input2")
   .addVertex("merge", new MergeVertex(), "L1", "L2")
   .addLayer("out", new OutputLayer.Builder().nIn(4+4).nOut(3).build(),
      "merge")
   .setOutputs("out")
   .build();

// Pass the configuration to a ComputationGraph
ComputationGraph computationGraph = new ComputationGraph(conf);
computationGraph.init()
```

Listing 3.11: Example of Configuring ComputationGraph

To load a multilayer network or computation graph from the local disk, the programmer just needs to specify the file locations of the networks and use the corresponding API:

```
// Load the MultiLayerNetwork
MultiLayerNetwork net2 = MultiLayerNetwork.load(new File(filePath), true);

// Load the ComputationGraph
ComputationGraph net2 = ComputationGraph.load(new File(filePath), true);
```

Listing 3.12: Example of Loading Network from Local Disk

Loading a pre-existing model from the DL4J ModelZoo is also easy. For example, the following code illustrates how a programmer instantiates a fresh, untrained network of AlexNet:

```
ZooModel zooModel = AlexNet.builder()
            .numClasses(numberOfClassesInYourData)
            .seed(randomSeed)
            .build();
Model net = zooModel.init();
```

Listing 3.13: Example of Loading Built-in Model

Some models have pretrained weights available, and a small number of models are pretrained across different datasets. *PretrainedType* is an enumerator that outlines different weight types, which includes *IMAGENET*, *MNIST*, *CIFAR10*, and *VGGFACE*.

For example, users can initialize a VGG-16 model with ImageNet weights like so:

```
ZooModel zooModel = VGG16.builder().build();
Model net = zooModel.initPretrained(PretrainedType.IMAGENET);
```

Listing 3.14: Example of Loading Built-in Model with Pretrained Weights

Due to the complexity of the realization models for the feature "Specify Network", the corresponding realization models are omitted.

3.4.4 Operate Network

As discussed in the section above, the APIs to run a *MultiLayerNetwork* or *ComputationGraph* are also different. To train a network, the corresponding network objects have the function *MultiLayerNetwork.fit(DataSet dataset)* or *ComputationGraph.fit(DataSet dataset)* (and other overloading functions) enabling the dataset to be fed into the models.

```
DataSet trainingData = data.getTrain();

// create and train a multilayer network for 1000 epochs
MultiLayerNetwork model = new MultiLayerNetwork(multilayerNetworkConf);
model.init();

for(int i=0; i<1000; i++ ) {
   model.fit(trainingData);
}

// create and train a computation graph for 1000 epochs
ComputationGraph computationGraph = new ComputationGraph(compGraphConf);
computationGraph.init();

for(int i=0; i<1000; i++ ) {
   model.fit(trainingData);
}
```

Listing 3.15: Example of Operating Networks

Figure 3.13 is the corresponding realization model containing the API elements in the above code snippet.

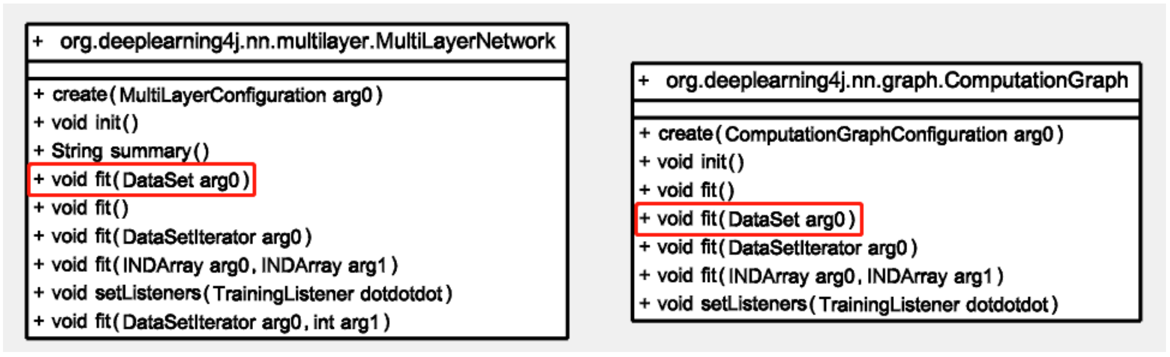

+ org.deeplearning4j.nn.multilayer.MultiLayerNetwork
+ create(MultiLayerConfiguration arg0)
+ void init()
+ String summary()
+ void fit(DataSet arg0)
+ void fit()
+ void fit(DataSetIterator arg0)
+ void fit(INDArray arg0, INDArray arg1)
+ void setListeners(TrainingListener dotdotdot)
+ void fit(DataSetIterator arg0, int arg1)
+ org.deeplearning4j.nn.graph.ComputationGraph
+ create(ComputationGraphConfiguration arg0)
+ void init()
+ void fit()
+ void fit(DataSet arg0)
+ void fit(DataSetIterator arg0)
+ void fit(INDArray arg0, INDArray arg1)
+ void setListeners(TrainingListener dotdotdot)

Figure 3.13: Realization Model of Training Networks (Partial)

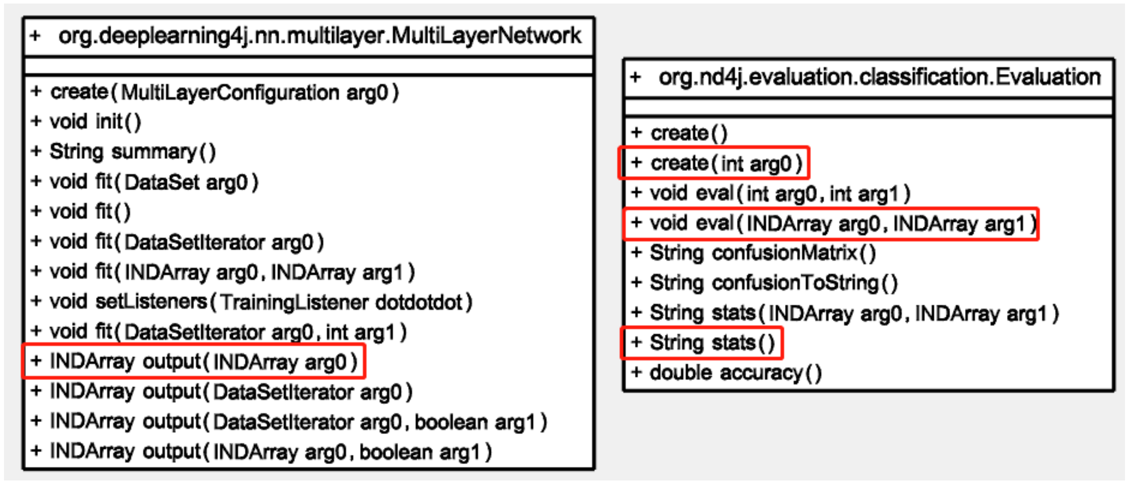
After the model is trained, the model output can be obtained using the function *MultiLayerNetwork.output(INDArray...)* or *ComputationGraph.put(INDArray...)* (and other overloading functions). With the model output and the ground truth labels, the model performance can be assessed by comparing the outputs during the evaluation phase in the following two ways.

The first way is using the specific class and pass the model output and ground truth as parameters. The package ND4J provides a class named Evaluation which calculates the accuracy, precision, Recall as well as improvement factor of the model. These metrics are obtained using the function *Evaluation.eval(INDArray realOutcomes, INDArray guesses)* (and other overloaded functions). The metrics can be accessed by printing *Evaluation.stats()* as follows:

```
//evaluate the model on the test set
Evaluation eval = new Evaluation(3);
INDArray output = model.output(testData.getFeatures());
eval.eval(testData.getLabels(), output);
System.out.println(eval.stats());
```

Listing 3.16: Example of Evaluating Networks

Figure 3.14 is the corresponding realization model containing the API elements in the above code snippet.



+ org.deeplearning4j.nn.multilayer.MultiLayerNetwork
+ create(MultiLayerConfiguration arg0)
+ void init()
+ String summary()
+ void fit(DataSet arg0)
+ void fit()
+ void fit(DataSetIterator arg0)
+ void fit(INDArray arg0, INDArray arg1)
+ void setListeners(TrainingListener dotdotdot)
+ void fit(DataSetIterator arg0, int arg1)
+ INDArray output(INDArray arg0)
+ INDArray output(DataSetIterator arg0)
+ INDArray output(DataSetIterator arg0, boolean arg1)
+ INDArray output(INDArray arg0, boolean arg1)
+ org.nd4j.evaluation.classification.Evaluation
+ create()
+ create(int arg0)
+ void eval(int arg0, int arg1)
+ void eval(INDArray arg0, INDArray arg1)
+ String confusionMatrix()
+ String confusionToString()
+ String stats(INDArray arg0, INDArray arg1)
+ String stats()
+ double accuracy()

Figure 3.14: Realization Model of Evaluating Networks (Partial)

In the feature "operate network", apart from the mandatory feature "run network", the optional feature "save network" encapsulates the APIs for storing the network on disk. The classes *MultiLayerNetwork* and *ComputationGraph* provide the corresponding functions named `save(File file)` to achieve this.

3.5 Feature Interactions

In the simplest case, features are independent of each other. In such a case, each feature corresponds to a set of API elements as explained in the previous section. Selecting two independent features then simply requires to compute the union of the corresponding realization models that contain the API elements for each of the features.

However, the features we specified for DL4J in the feature model shown in Figures 3.1 to 3.5 are at a high level of abstraction. Furthermore, we decided to structure the features according to the ML pipeline. As a result, the features are not always independent. There exists several so-called feature interactions, where the simultaneous presence of multiple features requires different or additional API elements to be used. For example, when the feature *transform data* is selected, the API elements that should be used depend on whether the feature *CSV data* or *image data* is also selected.

As described in section 2.1.2, CORE can deal with feature interactions by defining so-called conflict resolution models – realization models that realize a *set of features*. When the user selects features from the feature model, the TouchCORE tool will first check whether any conflict resolution models exist and prioritize those when composing the models.

In the scope of our DL4J feature model, there are 7 feature interactions and they will be illustrated in the following sections:

1. If the user selects more than one input data type, then an additional set of new API classes (e.g., *ConcatenatingRecordReader*,*ComposableRecordReader*) should be part of the API to integrate the different types of data. We created a conflict resolution model called *multiple data types* for these additional API elements.
2. The type of the input data chosen under *load data* changes the API for *structure data* and *transform data*.
3. The API for *load from disk* depends on the type of the network chosen, i.e., *multilayer network* or *computation graph*.
4. The API for *load built-in model* depends on the type of network chosen.
5. The API for *saving network* depends on the type of network chosen.
6. The API for *customizing high-level settings* depends on the type of network chosen.
7. The API for *operating network* depends on the type of network chosen.

3.5.1 Addressing Feature Interactions

To illustrate the complexity of the feature interactions, we show here how to address the first and the second feature interactions from the above list. The involved features are *CSV data*, *image data*, *structure data* and *transform data*. Since there is a cross-tree constraint that requires selecting *CSV data* or *image data* in case *structure data* or *transform data* is selected, there are $2^4 - 3 = 13$ different possible ways of selecting a subset of those features. Hence, it is laborious to enumerate these scenarios and build the corresponding realization models. There are some techniques to reduce the complexity.

3.5.1.1 Extracting Common Parts

A frequently used technique is extracting the common parts of the realization models to avoid repeatedly defining the same API model elements in each of them. Take the second feature interaction as an example. It happens between the features of "Load Data" and "Prepare Data". In our scope of abstraction, there are two options of data (i.e., CSV data and image data) that have corresponding data operations. The APIs of the feature "structuring data" and "transforming data"

		Load Data	
		CSV Data Type	Image Data Type
Prepare Data	Structure Data	Load and Structure CSV Data	Load and Structure Image Data
	Transform Data	Load and Transform CSV Data	Load and Transform Image Data

Table 3.2: Combinations of the "Load Data" and "Prepare Data" Feature Interaction

for these two data types are completely different. Hence, we have 4 combinations as illustrated in Table 3.2.

For the CSV data, even though structuring and transforming are different operations, there exist some intersections between their APIs. To elegantly build the realization models for these feature interactions, we extract their common parts and put those API elements in a more general model. Then the concrete realization models that need those API elements can extend this general model.

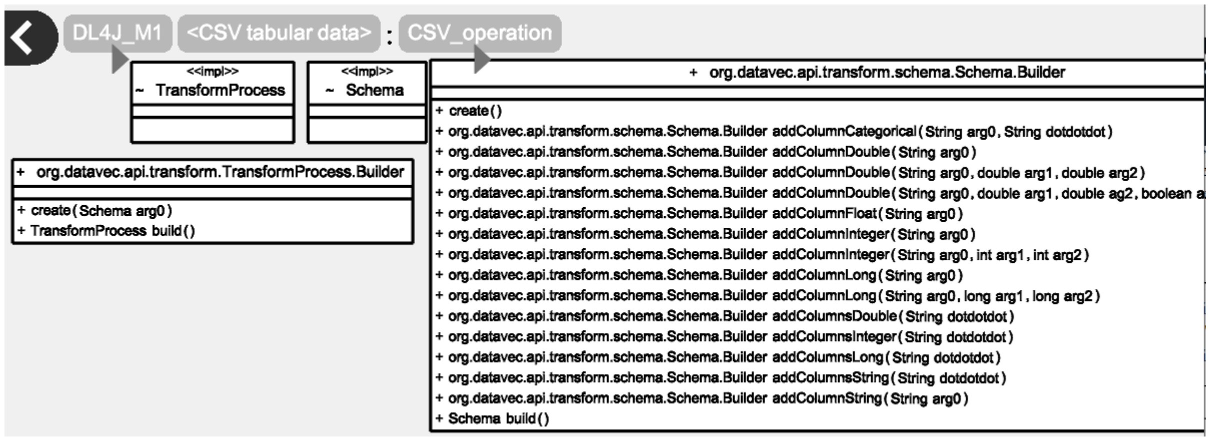
Below is a code snippet presenting the data operations on CSV data, where the brown color represents structuring and the blue color represents transforming.

```
Schema inputDataSchema = new Schema.Builder()
         .addColumnString("DateTimeString")
         .addColumnsString("CustomerID", "MerchantID")
         .addColumnDouble("TransactionAmountUSD",0.0,null,false,false)
         //$0.0 or more, no maximum limit, no NaN and no infinite values
         .build();

TransformProcess tp = new TransformProcess.Builder(inputDataSchema)
         .removeColumns("CustomerID","MerchantID")
         .conditionalReplaceValueTransform(
            "TransactionAmountUSD", /Column to operate on
            new DoubleWritable(0.0),
            new DoubleColumnCondition("TransactionAmountUSD",
            ConditionOp.LessThan, 0.0))
         .stringToTimeTransform("DateTimeString","YYYY-MM-DD HH:mm:ss.SSS",
            DateTimeZone.UTC)
         .removeColumns("DateTime")
         .build();
```

As we can see in the code snippet, to execute any data operation on the CSV data, the original schema of the CSV data needs to be specified first. Also, since all the data operations are using the class *TransformProcess.Builder*, the API *TransformProcess.Builder(Schema).build()* must be used at the end to complete building the data operation pipeline.

Hence, we built a general realization model for CSV data operation that contains the related class *Schema.Builder* and API *TransformProcess.Builder(Schema).build()* as shown in Figure 3.15.



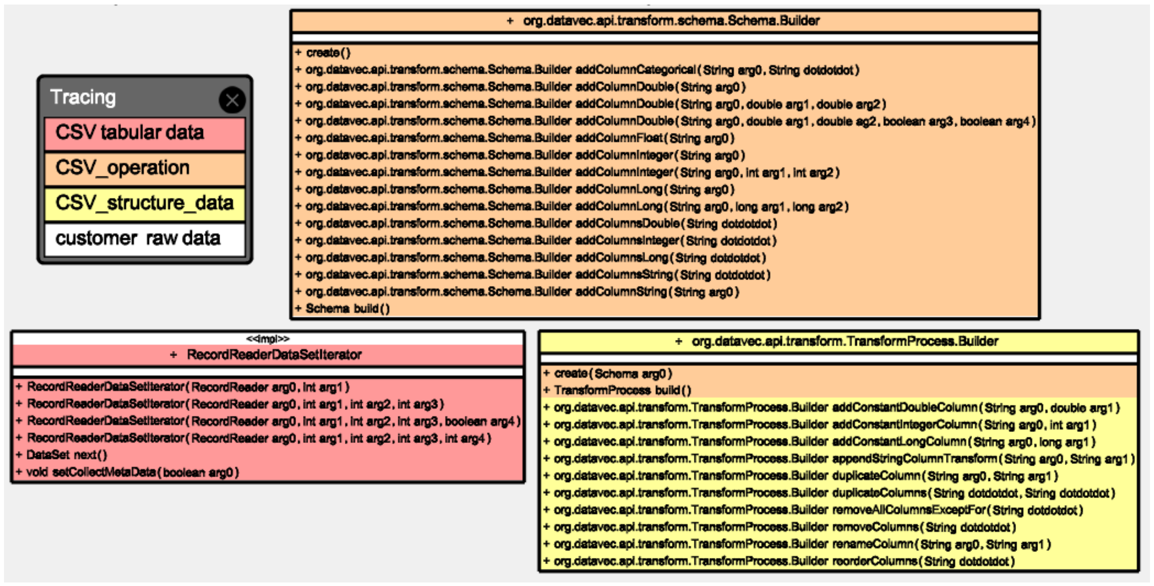
```
+ org.datavec.api.transform.TransformProcess.Builder
+ create(Schema arg0)
+ TransformProcess build()
```

```
+ org.datavec.api.transform.schema.Schema.Builder
+ create()
+ org.datavec.api.transform.schema.Schema.Builder addColumnCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0, double arg1, double arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0, double arg1, double ag2, boolean a
+ org.datavec.api.transform.schema.Schema.Builder addColumnFloat(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger(String arg0, int arg1, int arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong(String arg0, long arg1, long arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsDouble(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsInteger(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsLong(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsString(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnString(String arg0)
+ Schema build()
```

Figure 3.15: A General Realization Model for CSV Data Operations (Partial)

Then, we built the realization model for "structuring CSV data" and "transforming CSV data" respectively and extend this general realization model. Note that from the users' perspective, when they select "CSV data" and "structure data", they directly get the final result of "load and structure CSV data", we also include the "load CSV data" since it's necessary. All this happens transparently to the user.

Figure 3.16 shows the partial realization model after selecting "CSV data" and "structure data" with the tracing mode enabled. With tracing, the class diagram weaver assigns a color to every model element depending on which model it came from. In the figure, the red model elements relate to loading CSV data, the orange parts represent the general data operations on CSV data, and the yellow model elements are the concrete CSV structuring operations.



```
+ org.datavec.api.transform.schema.Schema.Builder

+ create()
+ org.datavec.api.transform.schema.Schema.Builder addColumnCategorical( String arg0, String dotdotdot )
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble( String arg0 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble( String arg0, double arg1, double arg2 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble( String arg0, double arg1, double ag2, boolean arg3, boolean arg4 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnFloat( String arg0 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger( String arg0 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger( String arg0, int arg1, int arg2 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong( String arg0 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong( String arg0, long arg1, long arg2 )
+ org.datavec.api.transform.schema.Schema.Builder addColumnsDouble( String dotdotdot )
+ org.datavec.api.transform.schema.Schema.Builder addColumnsInteger( String dotdotdot )
+ org.datavec.api.transform.schema.Schema.Builder addColumnsLong( String dotdotdot )
+ org.datavec.api.transform.schema.Schema.Builder addColumnsString( String dotdotdot )
+ org.datavec.api.transform.schema.Schema.Builder addColumnString( String arg0 )
+ Schema build()
```

```
<<impl>>
+ RecordReaderDataSetIterator

+ RecordReaderDataSetIterator( RecordReader arg0, int arg1 )
+ RecordReaderDataSetIterator( RecordReader arg0, int arg1, int arg2, int arg3 )
+ RecordReaderDataSetIterator( RecordReader arg0, int arg1, int arg2, int arg3, boolean arg4 )
+ RecordReaderDataSetIterator( RecordReader arg0, int arg1, int arg2, int arg3, int arg4 )
+ DataSet next()
+ void setCollectMetaData( boolean arg0 )
```

```
+ org.datavec.api.transform.TransformProcess.Builder

+ create( Schema arg0 )
+ TransformProcess build()
+ org.datavec.api.transform.TransformProcess.Builder addConstantDoubleColumn( String arg0, double arg1 )
+ org.datavec.api.transform.TransformProcess.Builder addConstantIntegerColumn( String arg0, int arg1 )
+ org.datavec.api.transform.TransformProcess.Builder addConstantLongColumn( String arg0, long arg1 )
+ org.datavec.api.transform.TransformProcess.Builder appendStringColumnTransform( String arg0, String arg1 )
+ org.datavec.api.transform.TransformProcess.Builder duplicateColumn( String arg0, String arg1 )
+ org.datavec.api.transform.TransformProcess.Builder duplicateColumns( String dotdotdot, String dotdotdot )
+ org.datavec.api.transform.TransformProcess.Builder removeAllColumnsExceptFor( String dotdotdot )
+ org.datavec.api.transform.TransformProcess.Builder removeColumns( String dotdotdot )
+ org.datavec.api.transform.TransformProcess.Builder renameColumn( String arg0, String arg1 )
+ org.datavec.api.transform.TransformProcess.Builder reorderColumns( String dotdotdot )
```

Figure 3.16: Partial Realization Model for Structuring CSV Data (Loading CSV Data is Included)

In the figure above, the yellow part and orange part share the same class *TransformProcess.Builder*. This is achieved by specifying a mapping between the different builder classes in the realization models.

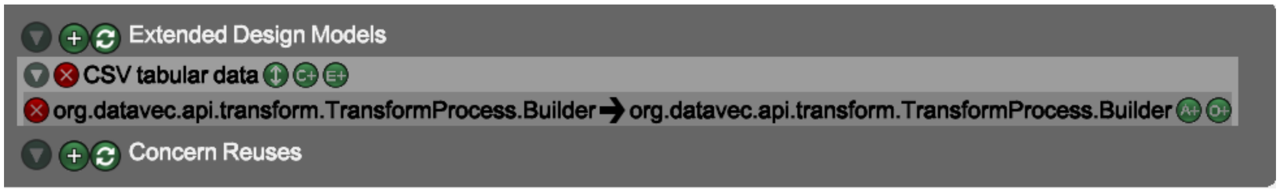
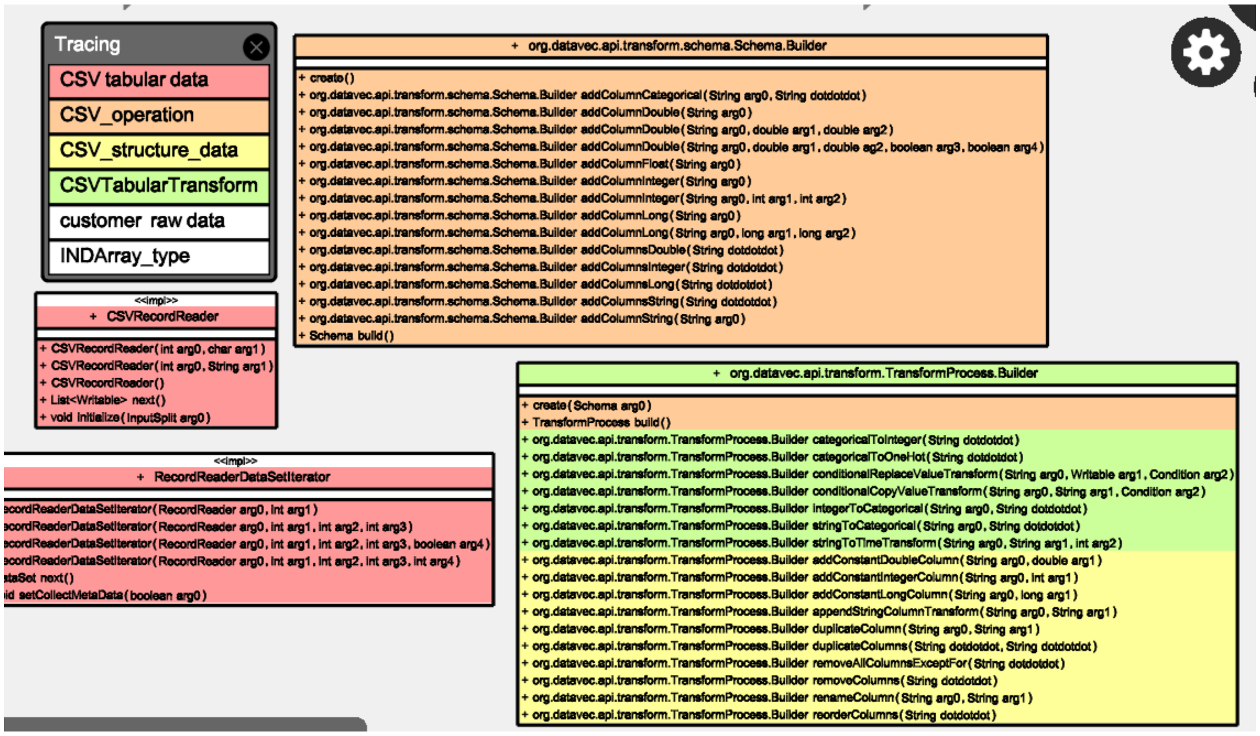

Extended Design Models
CSV tabular data
org.datavec.api.transform.TransformProcess.Builder → org.datavec.api.transform.TransformProcess.Builder
Concern Reuses

Figure 3.17: Class Mapping to Enable Merging

Considering the relation between "structuring" and "transforming" is not XOR, users can select them at the same time. In this scenario, the realization model result would integrate these two models and deal with the class mapping automatically. As shown in Figure 3.18, the API of the Builder class now contains methods coming from the CSV tabular transform realization model (green) as well as the CSV structure data realization model (yellow).



```
+ org.datavec.api.transform.schema.Schema.Builder

+ create()
+ org.datavec.api.transform.schema.Schema.Builder addColumnCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0, double arg1, double arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnDouble(String arg0, double arg1, double ag2, boolean arg3, boolean arg4)
+ org.datavec.api.transform.schema.Schema.Builder addColumnFloat(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnInteger(String arg0, int arg1, int arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong(String arg0)
+ org.datavec.api.transform.schema.Schema.Builder addColumnLong(String arg0, long arg1, long arg2)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsDouble(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsInteger(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsLong(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnsString(String dotdotdot)
+ org.datavec.api.transform.schema.Schema.Builder addColumnString(String arg0)
+ Schema build()
```

```
<<impl>>
+ CSVRecordReader

+ CSVRecordReader(int arg0, char arg1)
+ CSVRecordReader(int arg0, String arg1)
+ CSVRecordReader()
+ List<Writable> next()
+ void initialize(InputSplit arg0)
```

```
+ org.datavec.api.transform.TransformProcess.Builder

+ create(Schema arg0)
+ TransformProcess build()
+ org.datavec.api.transform.TransformProcess.Builder categoricalToInteger(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder categoricalToOneHot(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder conditionalReplaceValueTransform(String arg0, Writable arg1, Condition arg2)
+ org.datavec.api.transform.TransformProcess.Builder conditionalCopyValueTransform(String arg0, String arg1, Condition arg2)
+ org.datavec.api.transform.TransformProcess.Builder integerToCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder stringToCategorical(String arg0, String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder stringToTimeTransform(String arg0, String arg1, int arg2)
+ org.datavec.api.transform.TransformProcess.Builder addConstantDoubleColumn(String arg0, double arg1)
+ org.datavec.api.transform.TransformProcess.Builder addConstantIntegerColumn(String arg0, int arg1)
+ org.datavec.api.transform.TransformProcess.Builder addConstantLongColumn(String arg0, long arg1)
+ org.datavec.api.transform.TransformProcess.Builder appendStringColumnTransform(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder duplicateColumn(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder duplicateColumns(String dotdotdot, String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder removeAllColumnsExceptFor(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder removeColumns(String dotdotdot)
+ org.datavec.api.transform.TransformProcess.Builder renameColumn(String arg0, String arg1)
+ org.datavec.api.transform.TransformProcess.Builder reorderColumns(String dotdotdot)
```

```
<<impl>>
+ RecordReaderDataSetIterator

ecordReaderDataSetIterator(RecordReader arg0, int arg1)
ecordReaderDataSetIterator(RecordReader arg0, int arg1, int arg2, int arg3)
ecordReaderDataSetIterator(RecordReader arg0, int arg1, int arg2, int arg3, boolean arg4)
ecordReaderDataSetIterator(RecordReader arg0, int arg1, int arg2, int arg3, int arg4)
ataSet next()
id setCollectMetaData(boolean arg0)
```

Figure 3.18: Partial Realization Model for Structuring and Transforming CSV Data

For the image data, since there is no intersection in the "structuring" and "transforming" API, we did not need to define a common general realization model. The image "structuring" and "transforming" operations mainly rely on different classes like *CropImageTransform*, *FlipImageTransform*. Below are two figures for the different feature selections for image data.

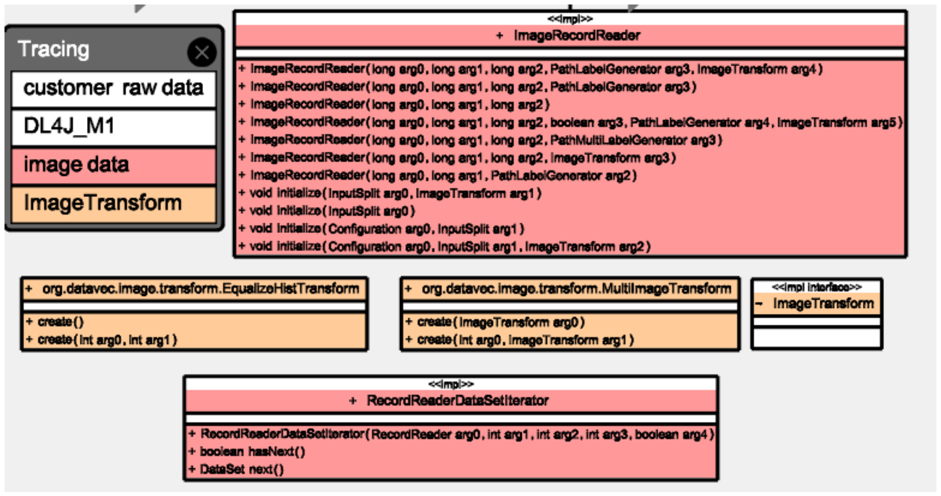

Tracing
customer raw data
DL4J_M1
image data
ImageTransform
<<impl>>
+ ImageRecordReader
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3, ImageTransform arg4)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2)
+ ImageRecordReader(long arg0, long arg1, long arg2, boolean arg3, PathLabelGenerator arg4, ImageTransform arg5)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathMultiLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2, ImageTransform arg3)
+ ImageRecordReader(long arg0, long arg1, PathLabelGenerator arg2)
+ void initialize(InputSplit arg0, ImageTransform arg1)
+ void initialize(InputSplit arg0)
+ void initialize(Configuration arg0, InputSplit arg1)
+ void initialize(Configuration arg0, InputSplit arg1, ImageTransform arg2)
+ org.datavec.image.transform.EqualizeHistTransform
+ create()
+ create(int arg0, int arg1)
+ org.datavec.image.transform.MultiImageTransform
+ create(ImageTransform arg0)
+ create(int arg0, ImageTransform arg1)
<<impl interface>>
~ ImageTransform
<<impl>>
+ RecordReaderDataSetIterator
+ RecordReaderDataSetIterator(RecordReader arg0, int arg1, int arg2, int arg3, boolean arg4)
+ boolean hasNext()
+ DataSet next()

Figure 3.19: Partial Realization Model for Transforming Image Data

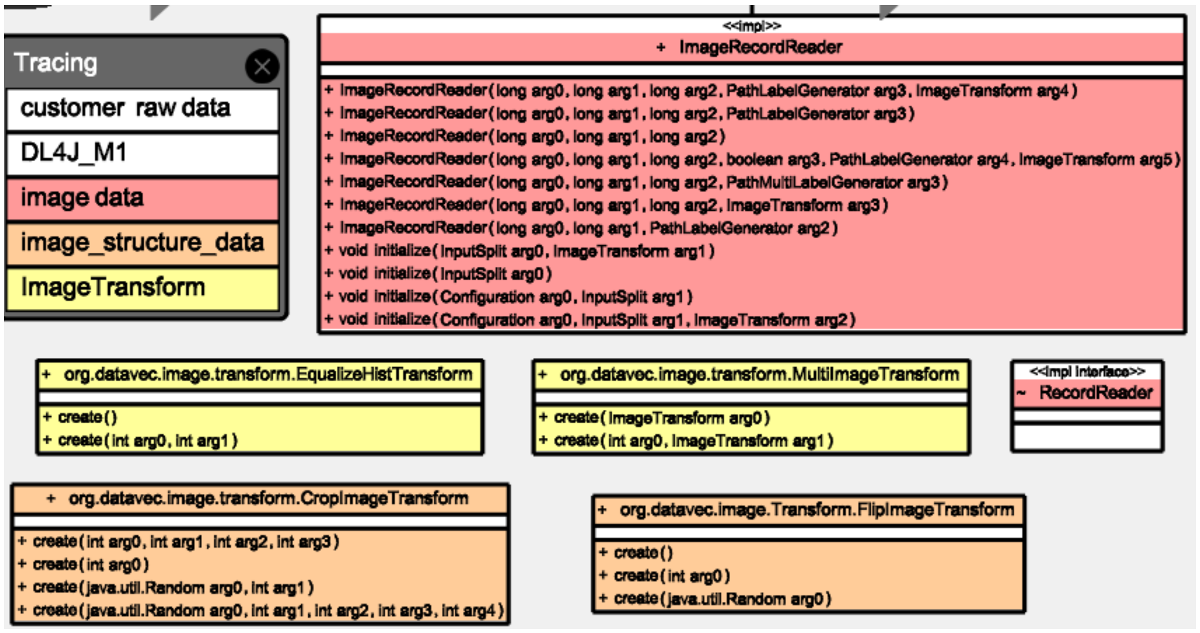

Tracing
customer raw data
DL4J_M1
image data
image_structure_data
ImageTransform
<<impl>>
+ ImageRecordReader
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3, ImageTransform arg4)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2)
+ ImageRecordReader(long arg0, long arg1, long arg2, boolean arg3, PathLabelGenerator arg4, ImageTransform arg5)
+ ImageRecordReader(long arg0, long arg1, long arg2, PathMultiLabelGenerator arg3)
+ ImageRecordReader(long arg0, long arg1, long arg2, ImageTransform arg3)
+ ImageRecordReader(long arg0, long arg1, PathLabelGenerator arg2)
+ void initialize(InputSplit arg0, ImageTransform arg1)
+ void initialize(InputSplit arg0)
+ void initialize(Configuration arg0, InputSplit arg1)
+ void initialize(Configuration arg0, InputSplit arg1, ImageTransform arg2)
+ org.datavec.image.transform.EqualizeHistTransform
+ create()
+ create(int arg0, int arg1)
+ org.datavec.image.transform.MultiImageTransform
+ create(ImageTransform arg0)
+ create(int arg0, ImageTransform arg1)
<<impl interface>>
~ RecordReader
+ org.datavec.image.transform.CropImageTransform
+ create(int arg0, int arg1, int arg2, int arg3)
+ create(int arg0)
+ create(java.util.Random arg0, int arg1)
+ create(java.util.Random arg0, int arg1, int arg2, int arg3, int arg4)
+ org.datavec.image.Transform.FlipImageTransform
+ create()
+ create(int arg0)
+ create(java.util.Random arg0)

Figure 3.20: Partial Realization Model for Structuring and Transforming Image Data

3.5.1.2 Reusing the Realization Models

From the previous section, we know that some realization models can be reused in different feature combinations. So we modularized the API elements related to those four features ("Load CSV Data", "Load Image Data", "Structure Data" and "Transform Data") into seven basic realization models, illustrated with orange boxes in the top row of Figure 3.21. To deal with feature interactions for all possibilities of selecting two features among the four we defined five conflict resolution models, shown in blue in the second row in Figure 3.21. These conflict resolution models extend the appropriate realization models shown in the first row. For example, the middle conflict resolution model in the second row deals with the case where both CSV data and image data is selected. In that case, the API should contain the API elements from *load CSV* and from *load image*, but also the ones contained in *multiple data*.

We also defined two conflict resolution models to deal with the case where three features are selected, namely both *CSV data* and *image data*, and either *structure data* or *transform data*. Those models, shown in the third row, extend the appropriate models in the second row.

There is no need to create a conflict resolution model for when all four features are selected, because the composition algorithm in that case will simply use *both* three-feature conflict resolution models.

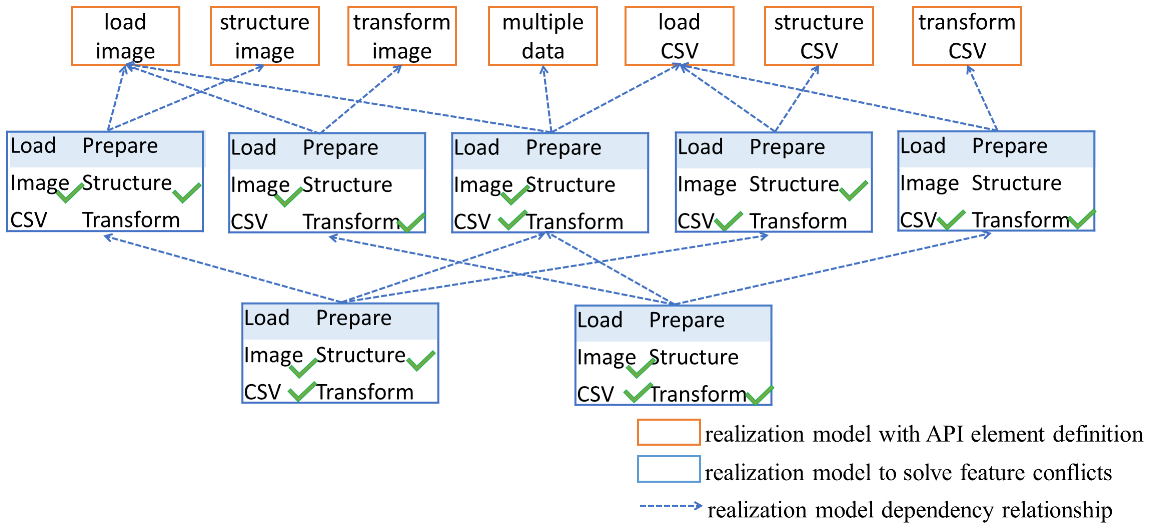

load image
structure image
transform image
multiple data
load CSV
structure CSV
transform CSV
Load
Prepare
Image
Structure
CSV
Transform
realization model with API element definition
realization model to solve feature conflicts
realization model dependency relationship

Figure 3.21: Manually Built Conflict Resolution Models for a Feature Interaction

The complexity of dealing with the feature interaction is of course hidden completely from the user of our approach. Figure 3.22 illustrates two feature selection combination examples and how our tool deals with generating the corresponding API.

In the first case shown at the top, *CSV data*, *image data* and *structure data* are selected. Our weaving algorithm finds the conflict resolution model linked to those 3 features (highlighted in purple) and then weaves all models in the extension hierarchy together, thus combining the APIs from *load CSV*, *load image*, *structure CSV*, *structure image* and *multiple data*.

In the second case, *CSV data*, *structure data* and *transform data* are selected. Our weaving algorithm finds two conflict resolution models, one linked to *CSV data* and *structure data*, the other one linked to *CSV data* and *transform data*, and therefore creates a new model (highlighted in purple) that extends these two models and then weaves all models in the resulting extension hierarchy together.

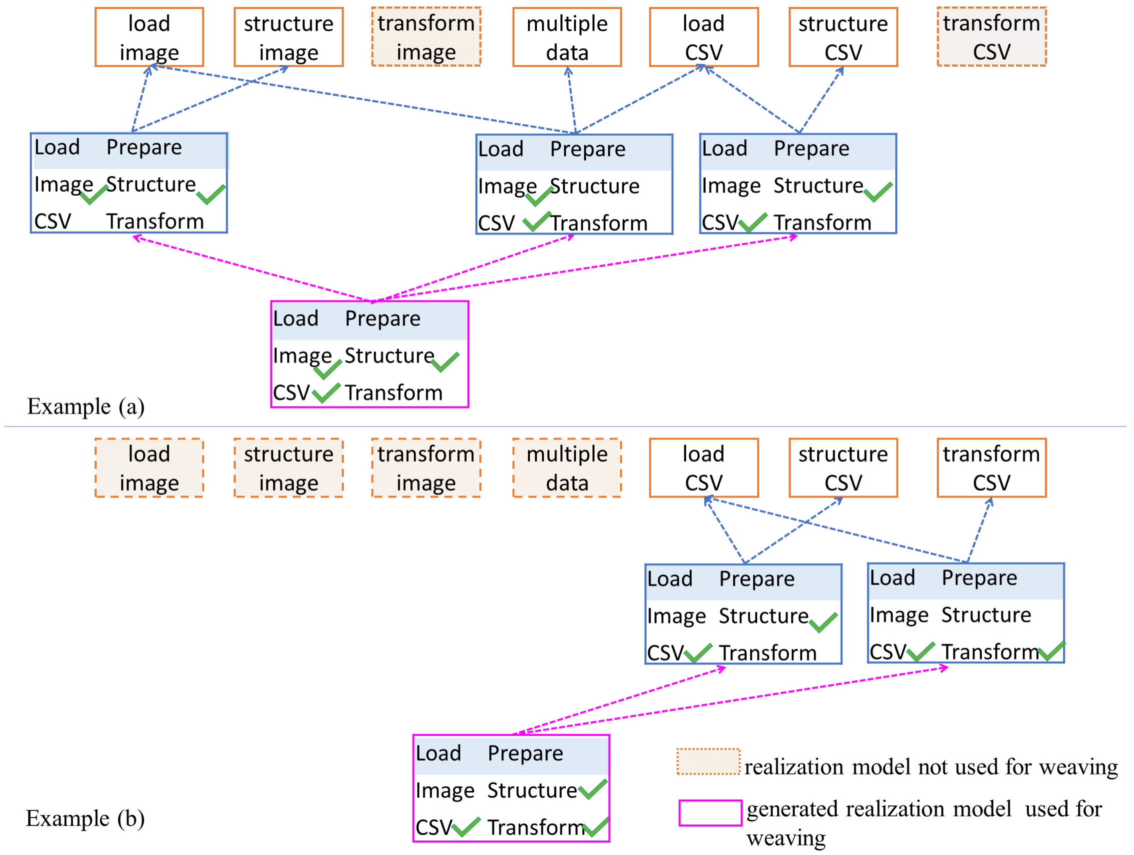


Figure 3.22: Examples of Returned Realization Models for Different Feature Combinations

To summarize, there exist feature interactions between the features of "load data" and "prepare data". We first investigated what classes and APIs are involved. Then we extracted the common parts (if any) as a general realization model. The corresponding realization models were built by extending the general part. We hide the complex feature interactions from users' views, enabling users to only focus on their application design.

The other 6 feature interactions have been dealt with in a similar way. So we will not present the realization details for them, but simply focus on their analysis.

3.5.2 Feature Interaction Analysis

In this section, we analyze the other feature interactions we had to address with some code snippets.

Network Type and Load Network From Disk In the feature of "load network from disk", the associated APIs depend on the type of the network the users want to load. Thus, the feature interactions are between "load from disk" and "MultiLayer Network", as well as "load from disk" and "Computation Graph" Even though the cases of the feature interactions are not complicated, DL4J provides two ways of each case.

```
// Load the MultiLayerNetwork
MultiLayerNetwork net1 = MultiLayerNetwork.load(new File(filePath), true);

// Load the ComputationGraph
ComputationGraph net3 = ComputationGraph.load(new File(filePath), true);

// or Load the MultiLayerNetwork using ModelSerilizer
MultiLayerNetwork net2 = ModelSerializer.restoreMultiLayerNetwork(new
   File(filePath));

// or Load the ComputationGraph using ModelSerilizer
ComputationGraph net4 = ModelSerializer.restoreComputationGraph(new
   File(filePath));
```

In the above code snippet, the blue code shows the first way of loading the network from disk while the brown code illustrates the second way. Hence, we also considered these two methods and provided them in our realization models. Because the involved APIs are not complex and there is no overlap between these two methods, we simply built the realization models for the feature interaction without creating a general common model.

Network Type and Load Built-in Network DL4J includes a model zoo so that users can use some well-known models directly instead of constructing them layer by layer. We manually classified these 16 provided models into the ones that are of type *MultiLayerNetwork* and the ones that are of type *ComputationGraph*. We use feature interaction models to ensure that when users select "ComputationGraph" and "Built-in Model", only those built-in models that are of type ComputationGraph are presented to the users. Similarly, another feature interaction model was created for the built-in networks that are of type multilayer network.

Network Type and Save Network Similar to loading, there are also feature interactions related to saving, i.e., between the feature "Save Network" and "Multilayer Network", as well as between "Save Network" and "Computation Graph". Again, DL4J provides two ways for each case, as illustrated in the following code snippet in blue and red.

```
// Save the MultiLayerNetwork
MultiLayerNetwork net1 = new MultiLayerNetwork(multilayerNetworkConfig);

net1.save(new File("path/to/save"))

// Save the ComputationGraph
ComputationGraph net2 = new ComputationGraph(computationGraphConfig);

net2.save(new File("path/to/save"))

// or Save the MultiLayerNetwork using ModelSerilizer
ModelSerializer.writeModelk(net1, new File("path/to/save"), true);

// or Save the ComputationGraph using ModelSerilizer
ModelSerializerwriteModelk(net2, new File("path/to/save"), true);
```

Network Type and its High-Level Settings The selection choices are: high-level settings for MultiLayerNetwork and high-level settings for ComputationGraph. Since there are many complicated APIs for the settings and there exists an overlap between the MultiLayerNetwork and ComputationGraph, we had to build a common general model. The details of these models are omitted here for space reasons.

Network Type and Run Network Since the number of choices of types of networks is 2 (multilayer and computation graph), and the number of choices of running a network is 3 (train, predict and evaluate), the number of combinations of those features is $2 \times 3 = 6$. In this case, 6 different feature interaction models had to be elaborated.

3.6 Reusing a Concernified Framework

The first four sections in this chapter explained how we concernified the DL4J library, i.e., built a feature model for it (section 3.3), specified realization models for each feature (section 3.4, and how we addressed the discovered feature interactions (section 3.5).

In this section, we explain by means of a code sample (see Appendix A) from the official DL4J example GitHub repository how our concernified DL4J library is intended to be used. By looking at the code comments, we first determine what features are involved. Then the process of generating the associated API is shown.

3.6.1 Determining the Used Features

After reading the code in Appendix A and inspecting the comments, we determined that this code sample intends to train and evaluate a double-layer MultiLayerNetwork on the built-in Mnist dataset.

Thus, the involved features are:

- **Load Data**: Load Built-in Data
- **Prepare Data**: Not involved
- **Specify Network**:
  - Network Type: MultiLayerNetwork
  - Construct Network:
    - Manually Build Network: Customize High-Level Settings
- **Operate Network**: Train, Predict and Evaluate

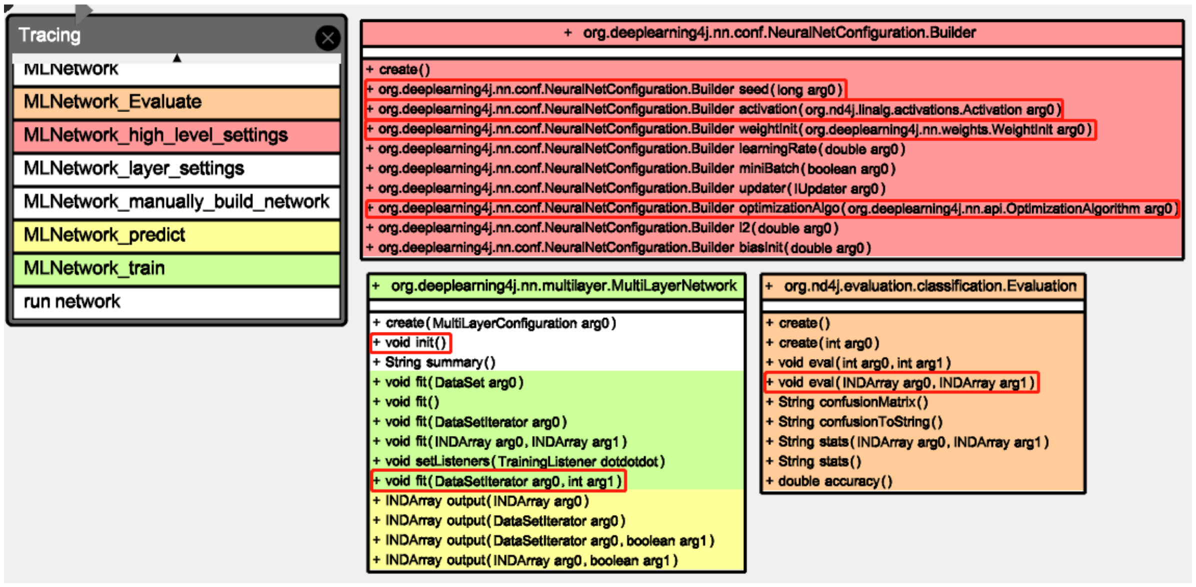
Even though feature interactions are hidden from the users, to prove the correctness of the concernified library, we still need to manually analyze the feature interactions and check their existence in the result.

Here, we have two feature interactions involved: “network type” and its “high-level settings”, and “network type” and “operate network”. So we can expect the following concrete feature interactions:

- **Network Type and its High-Level Settings**: High-Level Settings for MultiLayerNetwork
- **Network Type and Operate Network**: Train MultiLayerNetwork, Predict in MultiLayerNetwork, Evaluate MultiLayerNetwork

3.6.2 Reusing Process

As the feature model is supposed to be built from a beginner’s perspective, features are well grouped and clearly named. In general, users can directly use the feature model, selecting the features within the limitation of constraints.



```
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder

+ create()
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder seed(long arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder activation(org.nd4j.linalg.activations.Activation arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder weightInit(org.deeplearning4j.nn.weights.WeightInit arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder learningRate(double arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder miniBatch(boolean arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder updater(IUpdater arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder optimizationAlgo(org.deeplearning4j.nn.api.OptimizationAlgorithm arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder l2(double arg0)
+ org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder biasInit(double arg0)
```

```
+ org.deeplearning4j.nn.multilayer.MultiLayerNetwork

+ create(MultiLayerConfiguration arg0)
+ void init()
+ String summary()
+ void fit(DataSet arg0)
+ void fit()
+ void fit(DataSetIterator arg0)
+ void fit(INDArray arg0, INDArray arg1)
+ void setListeners(TrainingListener dotdotdot)
+ void fit(DataSetIterator arg0, int arg1)
+ INDArray output(INDArray arg0)
+ INDArray output(DataSetIterator arg0)
+ INDArray output(DataSetIterator arg0, boolean arg1)
+ INDArray output(INDArray arg0, boolean arg1)
```

```
+ org.nd4j.evaluation.classification.Evaluation

+ create()
+ create(int arg0)
+ void eval(int arg0, int arg1)
+ void eval(INDArray arg0, INDArray arg1)
+ String confusionMatrix()
+ String confusionToString()
+ String stats(INDArray arg0, INDArray arg1)
+ String stats()
+ double accuracy()
```

Figure 3.23: Result Model (Partial) for the Sample

In this example, users can select the features above and then get the realization model result. Users can also see the list of the involved features and trace the corresponding classes of each involved features in result model.

By dynamically interacting with the model which successfully concernifies the library, users can get the API recommendation based on their needs and it's very simple to make a change on their selections. As presented in the figure above, the APIs used in the sample code are contained. For example, the `NeuralNetConfiguration.Builder.seed(long arg0)` for the MultiLayerNetwork high level settings; `MultiLayerNetwork.fit(DataSet arg0)` for MultiLayerNetwork training; class `Evaluate` for MultiLayerNetwork evaluation.

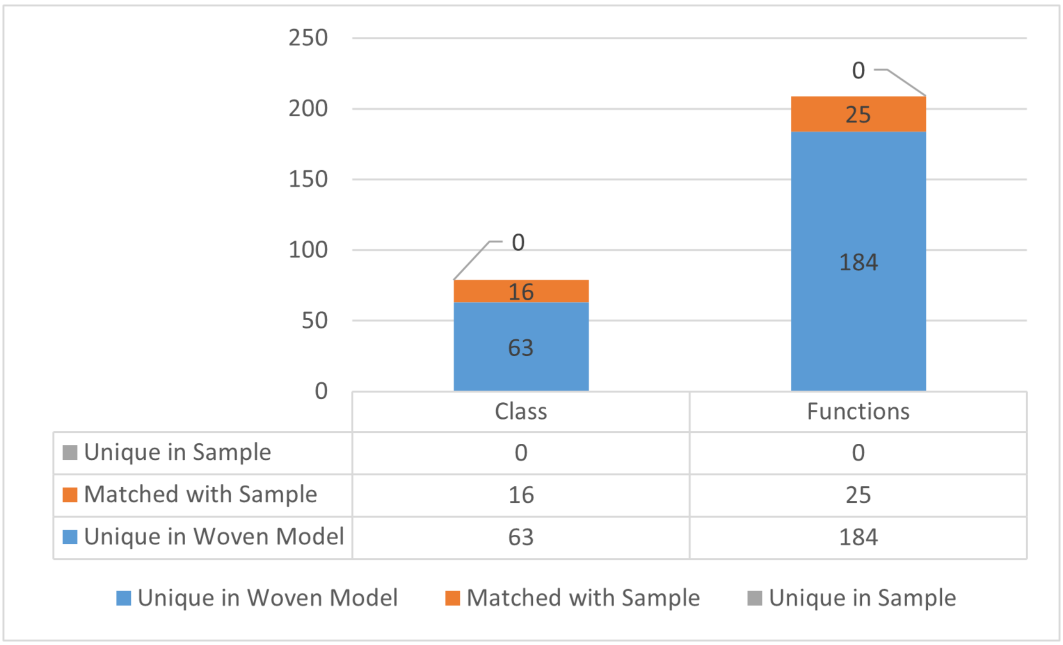
Within several minutes of selecting features, beginners already can get the classes and APIs might involved for their specific needs. Compared to the traditional learning path requires beginners to search in the Javadoc and consider the impacts resulting from conflicts of their choices, this way can significantly reduce users' exploring time. Compared to the traditional Javadoc, the feature model can explain its main usages in a well-structured way, which enables users to have an image of the library's functions in a glance.

3.6.3 Validity Check

The previous section explains the reusing process of the concernified framework, illustrating the simplicity and convenience of exploring the DL4J API by features. However, while convenience is desirable, correctness of the generated API is even more crucial. To check the correctness of the concernified DL4J library, we built a tool that examines whether the classes, APIs and constructors (called functions in the figure) used in the code sample are all present in the generated API. The results for the sample code are shown in Figure 3.24. The tool will be introduced in more detail in the next chapter.

The classes, APIs and constructors generated in the woven model are listed in Appendix (section B). To better present the matched ones, they are marked as brown color.

The recall rate of each category (class, API, constructor) for this sample is 100%, i.e., none of the used API elements were missing.



	Class	Functions
■ Unique in Sample	0	0
■ Matched with Sample	16	25
■ Unique in Woven Model	63	184

Figure 3.24: Recall Status of the Woven Model

4 Concernified Framework Quality Assessment

This chapter focuses on validating the quality of our concernified framework. To quantifiably measure the quality of the concernified *DL4J* framework, we conducted two experiments where we compare sample code that uses the Dl4J API with the suggested API elements generated by our approach.

Section 4.1 describes the validation metrics used as well as the validation workflow and the tools we built to automate the validation. Section 4.2 shows the results of the first experiment. We evaluate the performance of our concernified framework on those code samples that were used to build the framework to demonstrate that we built it correctly. Section 4.3 presents the second experiment, which evaluates how well our concernified APIs perform on new sample code. This provides an understanding of how well our approach would perform when used by a beginner who is writing a new application.

4.1 Experimental Setup

4.1.1 Assessment Metrics

To quantitatively check the quality of our concernified framework, we calculate two standard metrics, i.e., *recall* and *precision*, on a set of code examples from the official DL4J example github repositories. Since in our concernification we focussed on the API that is going to be used by beginners to build a complete machine learning pipeline, some advanced functionalities of DL4J are not included in our concernified API. As a result, we could not use all the code examples available in the github repository, but only used the samples in the data-pipeline-example folder[1] and dl4j-example folder[2].

The recall rate judges the coverage or correctness of the concernified framework, because it

[1]https://github.com/eclipse/deeplearning4j-examples/tree/master/data-pipeline-examples
[2]https://github.com/eclipse/deeplearning4j-examples/tree/master/dl4j-examples

measures whether all API elements that the code sample uses are actually present in our generated API. The precision measures how useful our approach is, as it calculates the ratio between the number of API elements the code sample actually uses vs. the number of API elements that our approach shows to the user, i.e., that are generated based on the high-level feature selection.

We used the standard confusion matrix[3] to define recall and precision metrics. Selecting features from the feature model can be considered a "search", and the generated API classes, functions are the result of the search. In that light, the whole procedure of reusing the DL4J concern can be seen as information retrieval.

Each code sample contains the set of API elements that a beginner programmer has used to achieve some specific functionality with DL4J. This set of API elements therefore represents the ground truth. We call it $S_{GroundTruth}$. Using our approach, the beginner programmer would select the high-level features they are interested in, and our tool would generate a set of API elements that the user should consider using. We call that generated set of API elements S_{Gen}.

By comparing $S_{GroundTruth}$ and S_{Gen} we get the 4 categories as indicated in Table 4.1.

	In S_{Gen}	Not in S_{Gen}
In $S_{GroundTruth}$	Relevant and Retrieved	Relevant and Not Retrieved
Not in $S_{GroundTruth}$	Non-Relevant and Retrieved	Non-Relevant and Not Retrieved

Table 4.1: Confusion Matrix for Generated API Validation

The intersection of the API elements found in the ground truth and those found in the API generated by our approach is the category of "Relevant and Retrieved" or often also called true positives. We call this set S_{TP} (True Positive).

$$S_{TP} = S_{GroundTruth} \cap S_{Gen} \tag{4.1}$$

The difference between $S_{GroundTruth}$ and S_{TP} is the category of "Relevant and Not Retrieved", labeled as S_{FN} (False Negative).

$$S_{FN} = S_{GroundTruth} - S_{TP} \tag{4.2}$$

The difference between S_{Gen} and S_{TP} is the category of "Non-Relevant and Retrieved", labeled as S_{FP} (False Positive).

[3]https://en.wikipedia.org/wiki/Confusion_matrix

$$S_{FP} = S_{Gen} - S_{TP} \tag{4.3}$$

The calculation of the *recall* and *precision* is calculated as follows:

$$Recall = \frac{|S_{TP}|}{|S_{GroundTruth}|} \tag{4.4}$$

$$Precision = \frac{|S_{TP}|}{|S_{Gen}|} \tag{4.5}$$

Besides the recall rate and precision rate, we propose another metric measuring the improvement factor that our approach provides over not using it, i.e., how many fewer API elements our generated API contains compared to the original DL4J API.

As mentioned already, DL4J is a giant library with a large API. Even when focussing only on the API useful for beginners as we do in this thesis, the DL4J_CORE.jar still contains 5542 classes. The improvement factor metric we propose to use calculates the reduction factor of the search space. In other words, it measures how much fewer API elements the beginner needs to consider when writing their code. The improvement factor is calculated as follows:

$$ImprovementFactor = \frac{|S_{DL4JAPI}|}{|S_{Gen}|} \tag{4.6}$$

4.1.2 Assessment Workflow and Tools

Figure 4.1 illustrates our validation workflow. Starting from a code sample (bottom left) we performed a coarse inspection of the code to determine the high-level features that are used. We select those features in the feature model we created for DL4J. Based on that selection, the TouchCORE tool takes care of feature interactions, and then combines the realization models associated with the selected features to generates a woven model (realization model result) containing the corresponding API elements, i.e., classes, methods and constructors.

To compare the API elements in this model with the ones used in the code sample we wrote three utility tools. The first one, the result converter, was implemented as a plugin to TouchCORE. It first finds the classes, methods and constructors. Then executes set addition in methods and constructors to create a function set: $S_{functions} = S_{methods} + S_{constructors}$. These classes and functions are exported into a plain text file (result).

The second tool (*GroundTruth Generator*) uses the *JavaParser* library to find the classes, methods, and constructors from the DL4J API that are used in the code sample and puts classes and functions into another text file (groundtruth). To make comparing easily, we intentionally use the same order for listing the found classes, functions in both files. The last tool, *Comparator Analyser*, is a Python script that compare the `result` and `groundtruth` text files, calculating the intersection, as well as the recall, precision and improvement factor metrics.

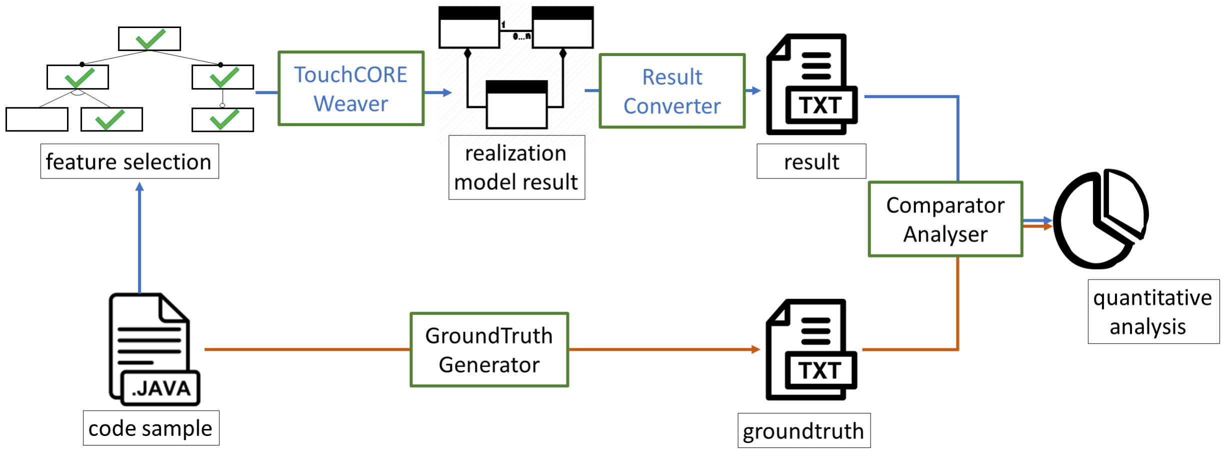


Figure 4.1: Authenticity Examining Process

4.1.3 Result Converter

Based on a feature selection, TouchCORE generates the corresponding realiation model that contains all the API elements corresponding to the selection. That model is a UML class diagram. This graphical interface allows beginner users to quickly get an overview of the structure of the API. Furthermore, the user can interact with the API, highlighting all the API elements corresponding to a feature in a certain color. To automate the validation tough, we have to convert the UML diagram into text format.

To achieve this, the extended the export functions of TouchCORE and added a function that iterates through all the classes in the current UML class diagram. For each class, the collection of its methods is inspected, and for each method, its signature is elaborated by concatenating the return type, the method name, and all the parameters. We use the fully qualified name of the class to avoid name clashes, and store all that information in a HashMap to avoid duplication.

In some cases, besides the classes, the generated result model also includes enumeration types. These enumerations are indispensable since they are parameters of certain methods existing in the woven model. When encountering this situation, we also record the fully qualified name of these enumerations. They are also stored in the same HashMap with the classes.

After the iteration of the UML diagram is completed, a text file is created. The classes and enumerations are output first, followed by the behavioral functions. Each section is partitioned by a specific marker illustrating the beginning and ending of that section. Below is the pseudocode of this woven model result converter.

```
initialize classes to be an empty set
initialize functions to be an empty set

for each entity that is a Class or Enumeration{
   store entity name in classes
   if entity is a Class{
      for each method of entity method{
         store method signature in the functions
      }
      for each constructor of entity constructor{
         store constructor signature in the functions
      }
   }
}
output the classes to result text
output the functions to result text
```

4.1.4 Ground Truth Generator

To create the groundtruth text file to compare against, we need to determine the classes and functions of the DL4J API used in the code sample. For this we are using a Java parser library called *JavaParser*.

4.1.4.1 JavaParser

JavaParser[4] is a Java library that allows users to interact with Java source code as Java objects representing an Abstract Syntax Tree (AST). The Java AST consists of several types of nodes, each containing some information known as structural properties. With this AST, users can traverse the tree and identify the pattern they are interested in, or manipulate the structure they found.

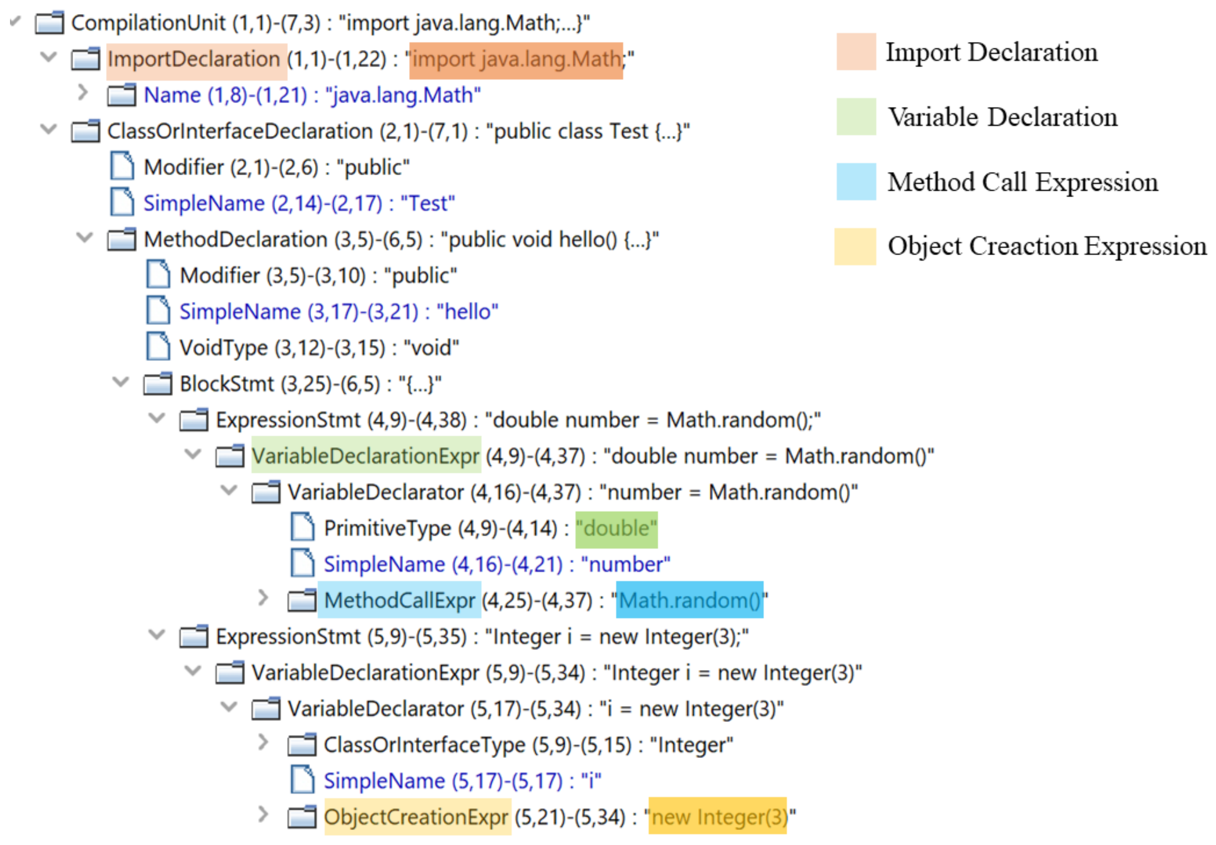
Below is a simple example of Java code. It starts with a package import statement, which can function as an indicator of what class is used in this code sample. The expression of *double number = Math.random();* contains the information of the variable type and what API is called. With this knowledge, we can infer the information of classes as well as methods called in the code sample.

```
import java.lang.Math;
```

[4]https://javaparser.org/

```
public class Test {
    public void hello() {
        double number = Math.random();
        Integer i = new Integer(3);
    }
}
```

Figure 4.2 shows the AST produced by JavaParser for that code sample, and the useful information that is used in our ground truth generator is highlighted in color.



```
CompilationUnit (1,1)-(7,3) : "import java.lang.Math;...}"
  ImportDeclaration (1,1)-(1,22) : "import java.lang.Math;"
    Name (1,8)-(1,21) : "java.lang.Math"
  ClassOrInterfaceDeclaration (2,1)-(7,1) : "public class Test {...}"
    Modifier (2,1)-(2,6) : "public"
    SimpleName (2,14)-(2,17) : "Test"
    MethodDeclaration (3,5)-(6,5) : "public void hello() {...}"
      Modifier (3,5)-(3,10) : "public"
      SimpleName (3,17)-(3,21) : "hello"
      VoidType (3,12)-(3,15) : "void"
      BlockStmt (3,25)-(6,5) : "{...}"
        ExpressionStmt (4,9)-(4,38) : "double number = Math.random();"
          VariableDeclarationExpr (4,9)-(4,37) : "double number = Math.random()"
            VariableDeclarator (4,16)-(4,37) : "number = Math.random()"
              PrimitiveType (4,9)-(4,14) : "double"
              SimpleName (4,16)-(4,21) : "number"
              MethodCallExpr (4,25)-(4,37) : "Math.random()"
        ExpressionStmt (5,9)-(5,35) : "Integer i = new Integer(3);"
          VariableDeclarationExpr (5,9)-(5,34) : "Integer i = new Integer(3)"
            VariableDeclarator (5,17)-(5,34) : "i = new Integer(3)"
              ClassOrInterfaceType (5,9)-(5,15) : "Integer"
              SimpleName (5,17)-(5,17) : "i"
              ObjectCreationExpr (5,21)-(5,34) : "new Integer(3)"
```

Figure 4.2: JavaParser Structural properties of an Example

With the information highlighted in figure 4.2, we can derive the following:

- Imported Class Names: all the packages / classes in the "import" statements point to classes that are being used by the sample code.

- Declared Variable Types: all the variable type related to *DL4J* (excluding primitive types) in the sample code.
- Complete Function Signatures: JavaParser provides ways to get the complete signatures of methods or constructors.

4.1.4.2 Ground Truth Generator Building Process

In our case, the code samples use mostly the related classes and functions from the DL4J and datavec library. Generally, JavaParser can only recognize the primitive types and contents in the default package (e.g., *java.lang*) if nothing provided for the type solver to look inside while solving types. To solve the problem, we provided the DL4J_CORE.jar file mentioned in the section 3.2 which we used to concernify the DL4J framework. We specify the JavaParser to look at the DL4J_CORE.jar file if it encounters some classes or functions out of the scope of primitive types and default package. Also, we filtered out those classes or functions which don't contain the keywords of "deeplearning4j", "nd4j" or "datavec". For example, the function `new String(data)` is excluded in the ground truth.

Like the woven model result needed to be tested, the ground truth would also have two categories of "classes", "functions". To get all the classes involved in the code sample, we took the import packages and declared variable types as the ground truth. All the methods and constructors retrieved by JavaParser are taken as the ground truth of "functions".

In the sample code, there might be cases where some method calls are made as part of an argument to another method call. For example, in the code of `trainDataSet.save(new File(trainFolder, "sample.bin"));`, the `File` constructor is called inside of the method call `DataSet.save(File arg0)`. Hence, for each method that the ground truth generator finds, the existence of any method calls happening inside the arguments should also be checked. To achieve this, we created a class `MethodCallVisitor` that extends the `VoidVisitorAdapter` and overrides its method `void visit(MethodCallExpr n, Void arg)`. Inside of this method, its super method `super.visit(MethodCallExpr n, Void arg)` is called to recursively retrieve all the method calls. And then all the methods are collected with a private field `apis` of `MethodCallVisitor`. To get the method calls caught by the visitor, a getter for this field is provided.

Below is the pseudocode of this ground truth generator.

```
initialize classesOfGt to be an empty set
initialize functionsOfGt to be an empty set

Process the code sample with JavaParser
```

```
Get the AST of the code sample

for each node of importdeclaration in the AST{
   add node name to classesOfGt
   }

for each node of variabledeclaration in the AST {
   add node name to classesOfGt
   }

for each node of methodcall in the AST {
   recursively get all the methods inside of the current method
   add their signatures to functionsOfGt
   }

for each node of objectcreation in the AST {
   recursively get all the constructors inside of the current object
      creation expression
   add their signatures to functionsOfGt
   }

output classesOfGt to groundTruth text
output the functionsOfGt to groundTruth text
```

4.1.5 Comparator and Analyser

In an experiment, there will be several code samples and their corresponding result and ground truth are stored in different folders. For example, for a code sample named `CSVDataModel.java`, its result is stored in `/result/CSVDataModel.txt` while the ground truth is in `/gt/` `↪CSVDataModel.txt`. The third tool we have is a Python script with the result path and ground truth path as input. The Python script first traverses these two folders and then compares the text files for a code sample.

For a code sample, the script will draw the Venn diagram for the category classes and functions separately. using the Equation 4.4, Equation 4.5 and Equation 4.6, its recall, precision rate and improvement factor can be obtained. When processing a code sample, the tool appends its recall, precision rate and improvement factor in corresponding arrays. After all the code samples are processed, the histogram of the recall and precision rate are drawn for further analysis.

Below is the pseudocode of this comparator.

```
initialize recalls to be an empty array
initialize precisions to be an empty array
initialize improvements to be an empty array

initialize resultpaths to be an empty array
initialize groundtruthpaths to be an empty array

for each result file in the result folder{
   append result file path to resultpaths
   }

for each groundtruth file in the groundtruth folder{
   append groundtruth file path to groundtruthpaths
   }

for i in length of results{
   read resultpaths[i]
   read groundtruthpaths[i]
   calculate recall, precision, and improvement
   append recall to recalls, precision to precisions, and improvement
      to improvements
   draw Venn diagram
   }

draw histogram based on recalls, precisions,
calculate the average improvement factor based on improvements
```

4.2 Validating Correctness

These section validates that we built our feature model correctly by evaluating the aforementioned metrics on the sample code that we used to build our concernified *DL4J* model. Because our model was built with these code samples, we expect the recall rate to be 100%. If that would not be the case, it would mean that we did a poor job in creating the realization models with the API elements and forgot to include some, or we mis-categorized some API elements and attached them to the wrong features. It also makes sense here to observe the precision rate. A higher precision rate indicates how relevant the generated API is. The improvement factor measures the ability to filter out as much of the irrelevant API as possible, hence, we want it to be as high as possible.

Since the code samples we collected came from the official GitHub repository of *DL4J*, some of them don't execute the whole machine learning pipeline (i.e., data loading, network building, and network running), but only a part of the pipeline to demonstrate some specific functionalities that *DL4J* provides. Our feature model was built for users who wants to perform a complete machine learning experiment, i.e., use the entire pipeline. When a user uses the *Feature Reuse* mode in our *TouchCORE* tool, the feature model constraints will ensure that the user makes a correct selection resulting in an API that supports the entire ML pipeline. We validate our feature model with the code samples that cover the entire pipeline in subsection 4.2.1.

We were also able to validate our model using the code samples which only cover a section of the machine learning pipeline. To do that we use the *Feature Construct* mode in our TouchCORE tool that makes it possible to bypass the feature model constraints when generating the API. We report on those results in subsection 4.2.2.

4.2.1 Validation for the Complete Machine Learning Pipeline Samples

We first show the details of two code samples – CVSDataModel and LinearDataClassifier – that both use the full machine learning pipeline. Figure 4.3 depicts two Venn diagrams that visually present the ground truth API set $S_{GroundTruth}$ (red) and the generated API set S_{Gen}.

Recall Rate: In these two code samples, for both *Classes* and *API* elements, the red circles are always entirely contained in the green circles. This means that all the relevant results are successfully retrieved. The *Recall* (see equation 4.4) for *Classes* and *API elements* in these 2 code samples is therefore 100% as expected.

Precision Rate: We can observe from the Figure 4.3 that our concernified *DL4J* model generates around 90 *Classes* and around 230 *Functions* for each of the full machine learning pipeline samples. Using equation 4.5, the *Precision* for category *class* for the CSVDataModel example is 20.4% and 21.7% for the LinearDataClassifier example. The *Precision* for category *API Elements* is 11.6% and 13.5%, respectively.

Improvement Factor: As explained in chapter 3, the number of public classes in *DL4J* relevant for beginners is 3741 and 92196 for public functions. The *Classes* improvement factor in the first sample is therefore $3741 \div 87 \approx 43$, whereas in the second sample is 45. The *Functions* improvement factor in the first sample is therefore $3741 \div 224 \approx 412$, whereas in the second sample is 415.

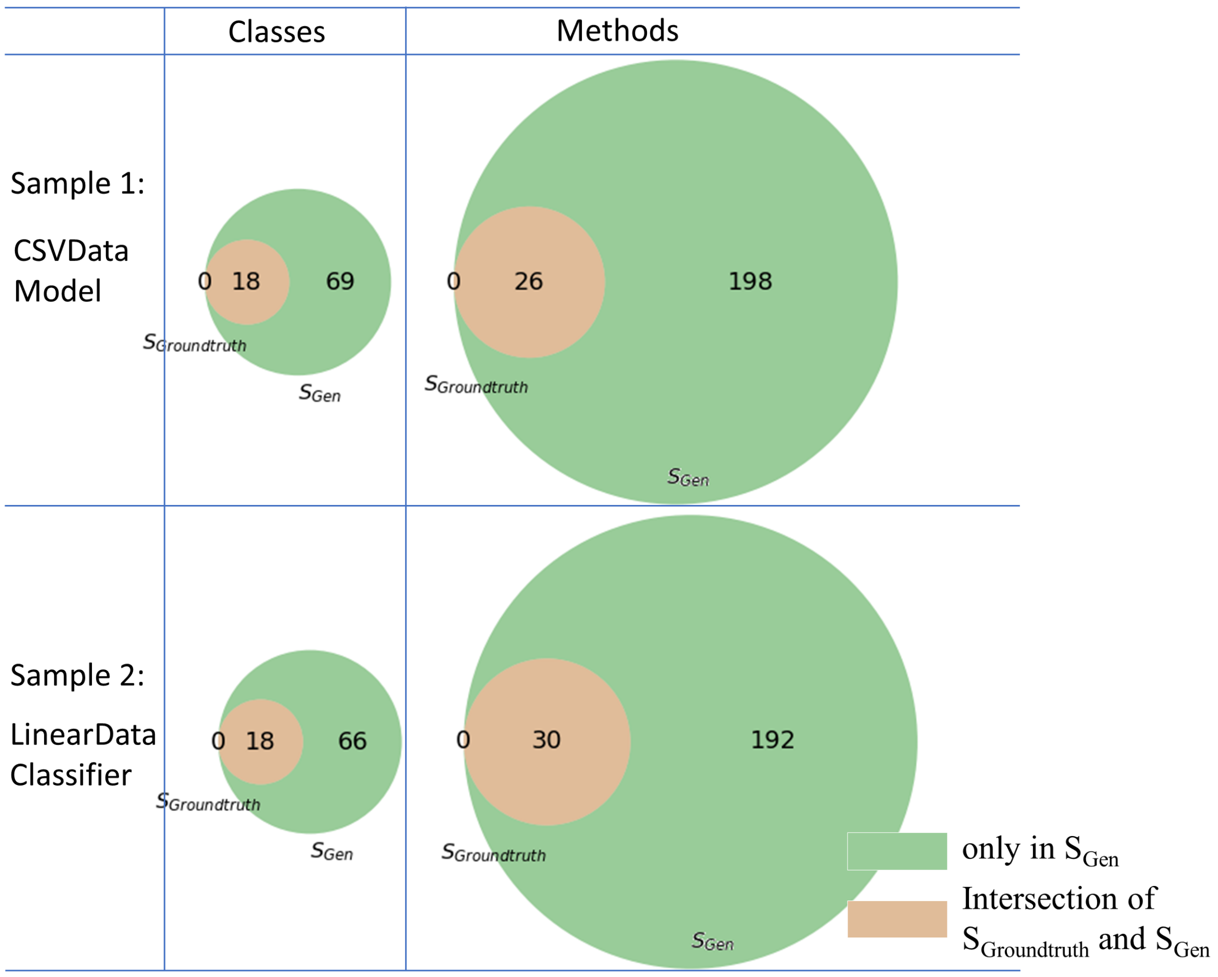


Figure 4.3: Detailed Validation Results for Two Full Machine Learning Pipeline Samples

Figure 4.4 shows a boxplot of the average recall and precision for the 13 complete ML pipeline validation code examples. As expected, the recall for both classes and API elements is 100%, hence our model was constructed correctly. The average precision for classes is 20.4%, whereas for API elements such as methods and constructors it is 13.5%.

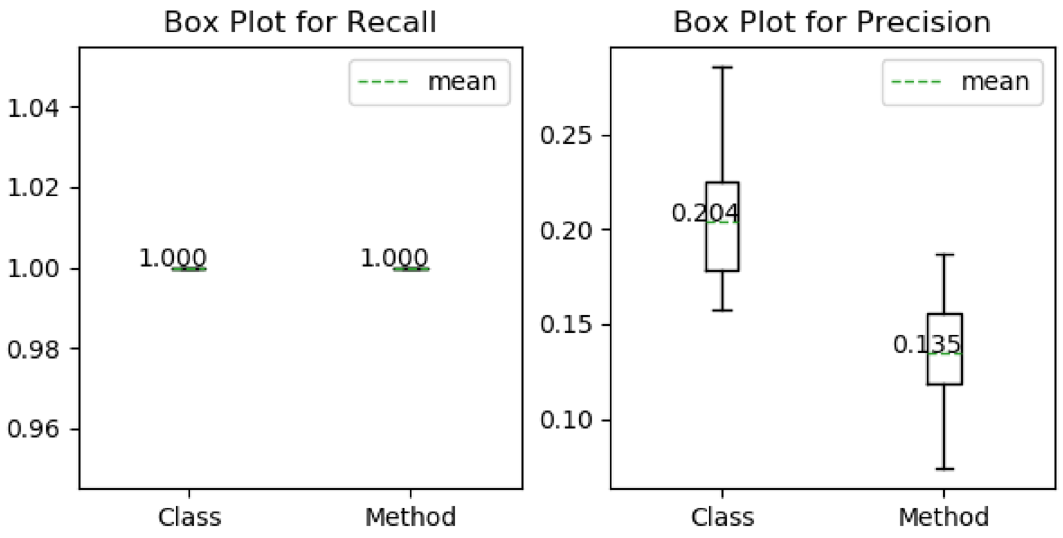


Figure 4.4: Recall and Precision for the Complete ML Pipeline Validation Samples

4.2.2 Validation for the Partial Machine Learning Pipeline Samples

Just as in the previous section, we show the detailed results for two code samples which only used parts of the machine learning pipeline. Their Venn diagrams are shown in Figure 4.5. The first sample loads one of the *DL4J* built-in datasets, partitions the data and then saves the partitioned data back to disk to demonstrate the functionalities of using a built-in dataset and saving a dataset. The second sample constructs a *MultiLayerNetwork*, saves it to disk, and loads the network from disk again to demonstrate the functionalities of building, saving and loading networks.

Recall Rate: These two code samples all have 100% *recall rate* for the *Class* and *API Elements* categories.

Precision Rate: Unlike in the full machine learning pipeline code samples, where our concernified *DL4J* model generated around the same number of *Classes* and *Functions* in most cases, the size of the ground truth and the generated API vary a lot more, mostly determined by the number of functionalities that the code samples cover. For example, the first code sample (FileSplitExample) only covers loading data files. Hence the ground truth API is very tiny (3 API elements are being used). The second code example (IrisNormalizer) uses two functionalities: load CSV data and transform data. Consequently it uses already a lot more API elements (10 in this case). Furthermore, some functionalities, e.g., constructing a network, can be achieved in several alternative ways using different sets of parameters. This results in our generated API containing in this case a lot more API elements than for other functionalities.

We can observe from the Figure 4.6 that the overall *Precision Rate* for the 7 partial pipeline samples is smaller than the complete pipeline samples, with an average of 13.1% for *Classes* and 7.2% for *Functions*. Besides, compared to the complete pipeline samples, the *Precision Rate* is more scattered.

Improvement Factor: Since the partial ML pipeline examples focus on a smaller part of *DL4J* than the complete ML pipeline examples it is not surprising that the improvement factor significantly higher. In the first sample, the *improvement factor* is $3741 \div 23 \approx 162$, while for the second sample it is $92196 \div 42 \approx 2195$.

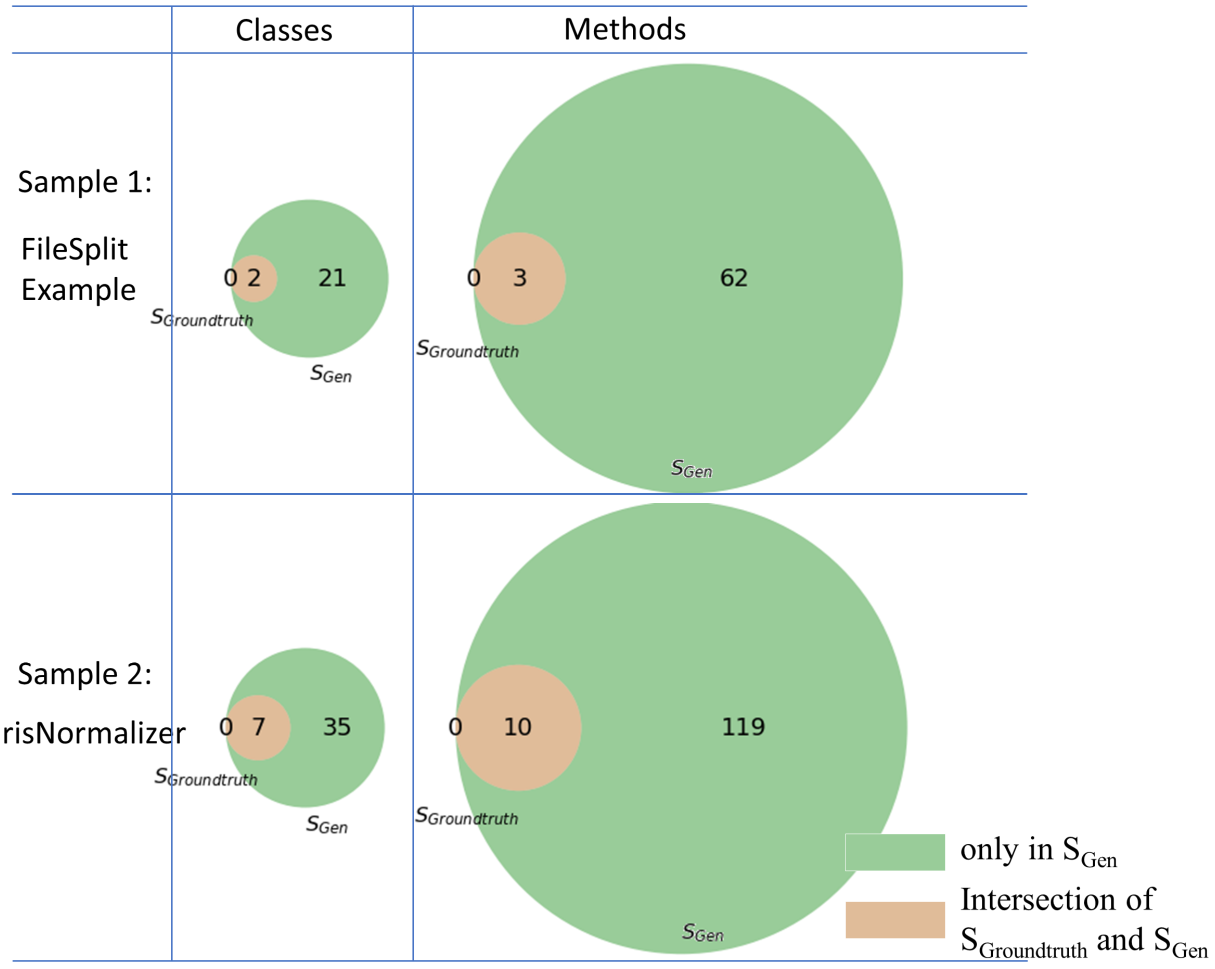


Figure 4.5: Detailed Validation Results for Two Partial Machine Learning Pipeline Samples

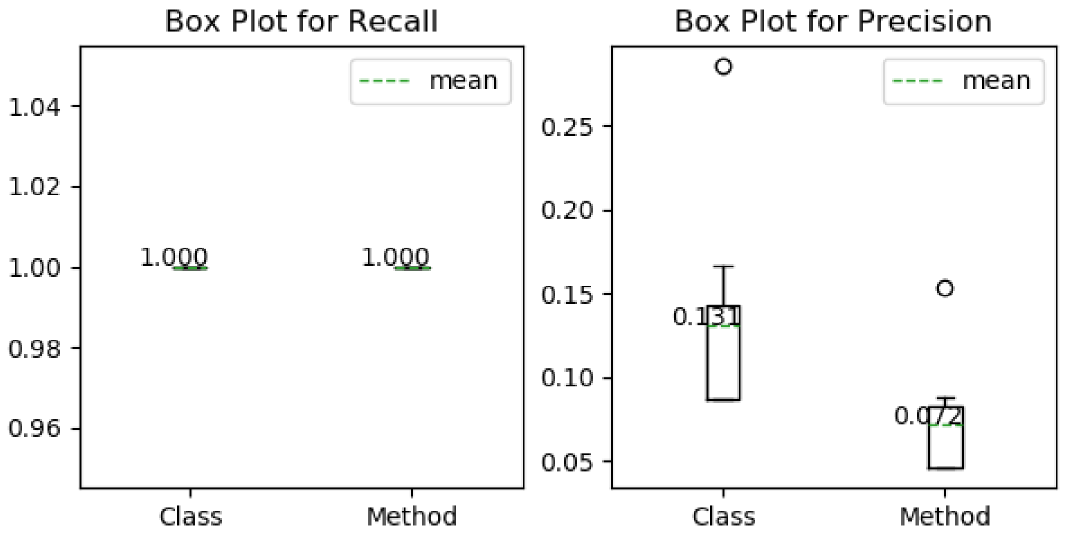


Figure 4.6: Recall and Precision for Partial ML Pipeline Validation Samples

Figure 4.6 shows the box plot measuring recall and precision for all of the 7 partial ML pipeline code examples. Again, the recall is 100%, and therefore our model was built correctly. The precision for the *Class* category is 13.1%, and for *Functions* it is 7.2%.

4.3 Evaluating Usefulness

While the previous section demonstrated that we correctly built our concern-oriented API for *DL4J*, we now want to evaluate whether our approach could simplify the task for beginners that are planning to write a new ML application from scratch. We do this by again determining the recall, precision and improvement factor metrics, but this time for code samples that have not been used to build our concernified API.

The code samples we chose for this purpose come from two public GitHub repositories. One is the official *DL4J* example repository [5]. The other one is the repository of a training course [6] which contains hands-on code for *DL4J* beginners attending the course. Since our concernified *DL4J* does not cover some advanced functionality and focuses only on the functionality that beginners tend to use, we did not use all the code examples in the repositories. We removed the more advanced ones, e.g., the one using Spark for distributed training.

[5]https://github.com/eclipse/deeplearning4j-examples
[6]https://github.com/CertifaiAI/cdle-traininglabs/tree/main/dl4j-labs/src/main/java/ai/certifai/solution

4.3.1 Evaluation of the Complete Machine Learning Pipeline Samples

In the end, we tested 14 complete new ML pipeline samples and achieved an average of 98.9% recall for API *Classes* and 98.0% recall for API *Functions*. The average precision is 19.3% and 13.8% for *Classes* and *Functions* respectively. Figure 4.7 shows the box plot and histogram for the recall and precision of our test samples. In the box plots, outliers are depicted as hollow circles, while averages are drawn as green lines. At the same time, the , the first quartile (Q1), the third quartile (Q3) and the maximum compose the main part of the box plot, and the length of the box gives an indication of the sample variability.

As the bottom-left histogram shows, there is one code sample achieving around 92% recall and two code samples achieving around 96% recall for *Classes*. Even though these three samples have different recall for *Classes*, they each failed in retrieving only one single class. We will discuss each of the examples in the following paragraphs.

Sample `CustomActivationUsageEx.java` has 12 classes as part of the ground truth. Our generated API did not contain the class `org.nd4j.linalg.activations.` `↪BaseActivationFunction`. This class was extended in the sample to demonstrate how to customize activation instead of using the ready-made activations. When doing our concernification, we decided that it would be simpler for beginners to use the predefined activation, and hence we exposed to the user the concrete activation enumeration `org.nd4j.linalg.activations.` `↪Activation`, which the user can directly pass as a parameter to different methods to select the activation function.

The samples `LoadCSVHousePrice.java` and `WomenChessPlayer.java` use the class `org.datavec.api.transform.condition.column.InvalidValueColumnCondition`, which was not in our generated API. This class is used to build a condition applied to a single column for filtering out the invalid values. For example, if the CSV dataset contains String values in an Integer column and these String values cannot be parsed into Integer, the `InvalidValue-ColumnCondition` would return `true` as these String values are invalid values according to the schema. Because this class is not included in the code samples we used to build our concernified DL4J model, the model does not contain this API. Instead, during our process of concernification, we found the Interface `org.datavec.api.transform.condition.Condition` is implemented by 17 types of concrete condition classes(including `InvalidValueColumnCondition`). Thus, we provided the Interface `Condition` in our DL4J model.

Compared to the recall rate of *Classes*, recall rates of category *Functions* vary between 93% and 100%. There are 6 samples having recall rate from 93% to 98% and the remaining 8 samples achieve 100% recall. For code sample `WomenChessPlayer.java`, which has the smallest recall in *Functions* (93.2%), our generated API did not contain 4 of the used functions out of a total of 59 used in the ground truth. The first missed function is `INDArray.shapeInfoToString()`. The code sample uses it to print out the dataset shape information, which allows beginners to get an idea of the dataset dimension. Because it is not indispensable in the machine learning pipeline, we had decided to not include it in the concernified API. The second missed function is `TransformProcess.Builder.stringMapTransform(String,Map<String,String>)`. It replaces one or more `String` values in the specified column with new values. Here again we had decided to not include this function in our API for beginners, as the same effect can be achieved by using simpler, standard Java methods. Finally, the third and fourth missed functions are `TransformProcess.Builder.integerColumnsMathOp(String,MathOp,String...)` and the constructor of `InvalidValueColumnCondition`. The function calculates and adds a new integer column to the dataset by performing a mathematical operation on a number of existing columns. This is clearly a complex operation and most likely would not be used by beginners, so again we decided not to include it when building the *DL4J* concern. The reason of missing the constructor of `InvalidValueColumnCondition` is explained in the previous paragraph.

The concernified *DL4J* model generated 92 classes and 243 functions for these 14 code samples on average. We can see from the bottom-right histogram in Figure 4.7 that precision rates of *Classes* concentrate in the range of 20% to 25%, and precision rates of *Functions* are mainly distributed between 10% and 15%. The overall precision rates for *Functions* are smaller than the precision rates of *Classes*. This makes sense, because a function might have several sets of parameters, which often translates into overloaded methods. Typically, a code sample however use only one of these overloaded functions depending on the specific needs.

The fact that the precision rates are not high does not mean our concern model performs poorly in retrieving relevant API elements. Even though the API elements returned by the model do not appear in the ground truth, some of them are still related to the ground truth API elements. In other words, some of the API elements in $S_{Gen} - S_{GroundTruth}$ can still provide knowledge to the users to some extent. Like the example `WomenChessPlayer.java` mentioned above, although the class `Condition` returned by the concern was not used in the ground truth, the users can get the ground truth class `InvalidValueColumnCondition` by searching `Condition` and find all the classes that implement this interface, including the ground truth class `InvalidValueColumnCondition`.

If this situation turns out to occur often, though, then it probably would be better to include all 17 concrete subclasses of the *Condition* interface.

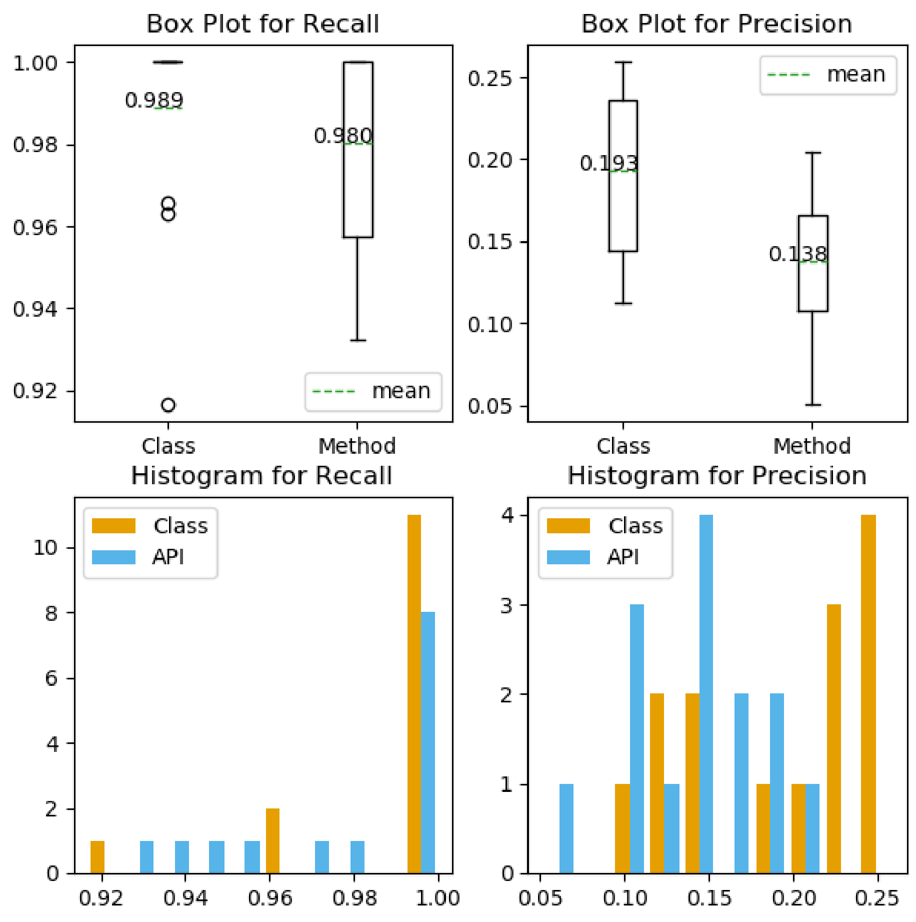


Figure 4.7: Recall and Precision for Complete ML Pipeline Evaluation Samples

To calculate the improvement factor for these complete machine learning pipeline samples, we first count the average number of generated API elements, which are 92 *Classes* and 243 *Functions*. Using the Equation 4.6, the improvement factor for *Classes* is $3741 \div 92 \approx 40$, whereas for *Functions* it is $92196 \div 243 \approx 379$. The improvement factor reflects the concern's ability to reduce the number of API elements of the library that are exposed to the users. In the experiment on complete machine learning pipeline samples, the API elements are decreased by one order of magnitude for *Classes* and by two orders of magnitude for *Functions*, meaning that the cognitive load on the users is greatly reduced. For example, searching through the reduced API will be much less effort.

4.3.2 Evaluation of the Partial Machine Learning Pipeline Samples

Although we did not design our concernified DL4J model for the incomplete machine learning pipeline, we still want to check out our concern's performance on those partial ML pipeline samples. Since the constraints in the concern reusing mode of TouchCORE enforce users to select features that build up a complete ML pipeline, we had to use the TouchCORE concern building mode to bypass these constraints. As shown in Figure 4.8, in the 7 partial ML pipeline samples that we evaluated we achieved an average of 94.6% recall rate for *Classes* and an average of 89.6% recall rate for *Functions*. However, the recall rate for functions varies among the samples, some of them achieved 100% while others only reached around 70%. This is caused by the fact that they only cover the incomplete ML pipeline as explained in the following paragraphs.

The samples that do not treat the entire ML pipeline contain a smaller numbers of features, narrowing down the scope. As a result, the feature interactions involved tend to also be simple. It turns out that in those situations the conflict resolution realization models are able to solve those feature interactions to achieve good recall. However, for those samples having relatively lower recall, they contain some APIs that belong to some features which are not suitable to be selected. Besides, since the incomplete ML pipeline has fewer features, the total number of the involved APIs would be small. Under such circumstances, the lost of APIs will have large impact on the falling of recall rate.

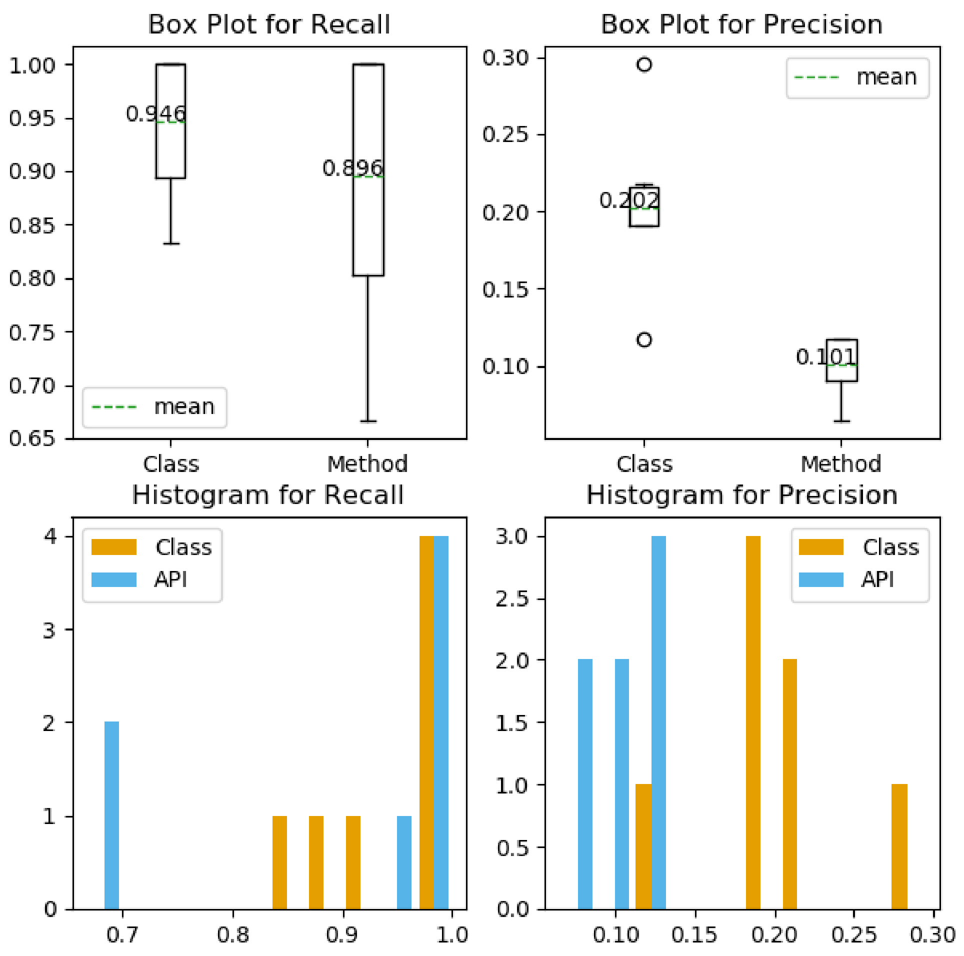


Figure 4.8: Recall and Precision for Partial ML Pipeline Evaluation Samples

Take the code sample `LoadCSV.java` which gets an unsatisfying recall rate (90% in *Classes* and 66.7% in *Functions*) as an example. It contains the procedures of loading CSV data, converting it into dataset and transforming the dataset. Besides these procedures, it also separates the dataset into training and testing datasets, which is the functionality comprised in the feature of "train" and "predict" whose APIs are mostly about running the network. Therefore, the features "train" and "predict" not selected during experiment led to the failure of retrieving the "separate dataset" APIs.

The APIs failed to be retrieved in our concernification *DL4J* for the sample `LoadCSV.java` are listed below and shown in Figure 4.9. For example, to better showcase the building process of dataset, the code sample uses `DataSet.getFeatures()` and `INDArray.shape()` sequentially (i.e., `DataSet.getFeatures().shape()`) to print out the shape of the training batch vector. As we can see from their signatures, the APIs are related to the unselected feature "train" and "predict". As we built the concern for the complete machine learning pipeline usage, some features are mandatory (like the feature "train" in this example). Thus, we put those APIs that can be classified into several features into some mandatory features to make sure they will definitely be included after the users make their feature selections. In the future, we could update our concern model for use when creating applications that only cover a part of the machine learning pipeline. In that case, we would have to change the feature constraints of the feature model, switching many mandatory features to optional ones. Furthermore, the API elements that are used in more than one feature would have to be assigned to all of them to ensure that the generated results will not miss these APIs.

In such case, the relatively low recall rates for incomplete machine learning pipeline samples are understandable.

```
//Class:
org.nd4j.linalg.dataset.SplitTestAndTrain

//Method:
org.nd4j.linalg.api.ndarray.INDArray.shape(),
org.nd4j.linalg.dataset.DataSet.getFeatures(),
org.nd4j.linalg.dataset.SplitTestAndTrain.getTest(),
org.nd4j.linalg.dataset.SplitTestAndTrain.getTrain(),
org.nd4j.linalg.dataset.DataSet.splitTestAndTrain(double)
```

In average, the concernified *DL4J* generated 38 *Classes* and 113 *Functions* for these experiment samples. The improvement factor for *Classes* is 98 and 815 for *Functions*, which are both higher than the statistics of the complete machine learning pipeline samples. Because the number of API elements generate by the concernified *DL4J* is considerably smaller than the original *DL4J* library, user can take advantage of it and try different feature combination if they do not find the elements they desire in the generated API.

From the previous sections, we can draw a conclusion in the statistics for the evaluating experiment:

- Generally, *Recall Rates* for the full or partial machine learning pipeline samples can reach 90% even 98%, meaning that most the relevant APIs are successfully retrieved in our concernified *DL4J* model. And it is understandable for those samples belong to incomplete pipeline that do not reach that level.

- *Precision Rates* for the full machine learning pipeline samples are stable, while for the partial pipeline samples, *Precision Rates* vary based on the functionalities the pipeline covers.

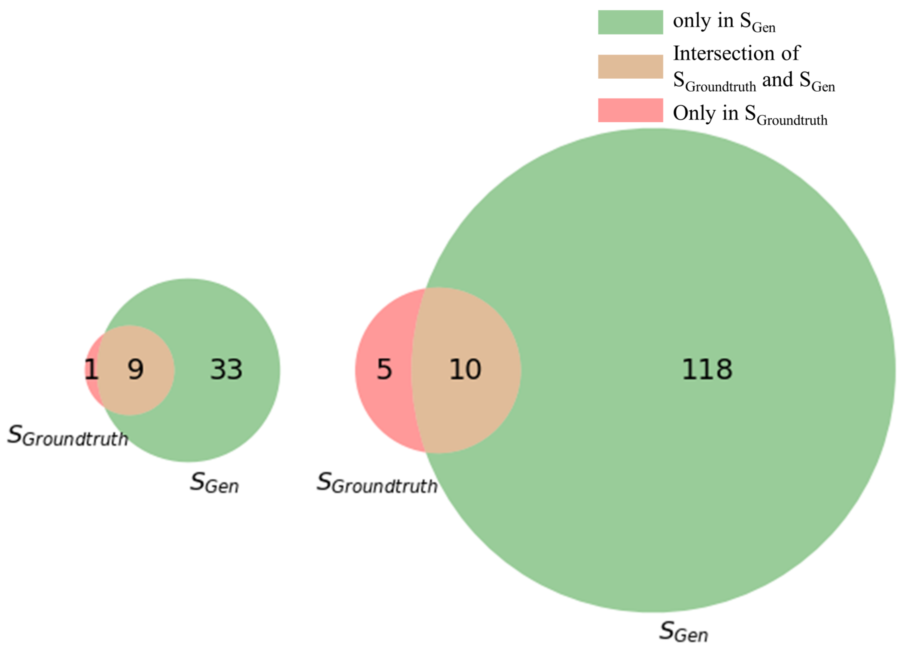


Figure 4.9: $S_{Groundtruth}$ and S_{Gen} in "LoadCSV" Sample

- *Improvement Factors* for the full machine learning pipeline samples are tend to be smaller than the partial pipeline samples since the partial pipeline samples contains less functionalities.

4.4 Analyzing the Results

On the API element level, the missed model elements in our generated API belong to one of two cases: 1) API elements belonging to more than one feature, or 2) alternative API elements. We explain each case below.

API Elements Belonging to more than one Feature: For example, `Nd4j.create(double[] data)` which creates an N-dimensional array, can initialize an array used as a dataset. Or it can also be used for constructing the array of the high-level parameters of a network. This specific API can therefore be used in different scenarios and thus belongs to multiple features. In our concernified API we had associated this constructor with the *Custom Data* feature, but not with the *Customize High Level Settings* feature. This shows that even with deep knowledge of the DL4J API and diligence when creating the feature model, omissions when assigning API elements to features can happen. Similarly, one could envision that some feature descriptions are not clearcut, and as a result some users interpret them differently from others. For example, should the preprocessing operations to convert a column from `String` to `int` for CSV data be classified under the feature *Structure Data* or *Transform Data*? Maybe it is appropriate to be classified into *Transform Data* because it changes the data range. But as it also changes the schema of the CSV data, it can also be categorized into *Structure Data*. Again it is probably wise to classify the operation under both features just make sure the API is being proposed to the user regardless of their interpretation of the categories of operations.

It is therefore important to allow the concernified API to be updated when such a case is detected.

Alternative API Elements: In DL4J, some functionality can be achieved in different ways, i.e., by using a different sets of APIs. In these cases, to simplify the task of learning DL4J for newcomers, we decided to provide the simplest or most frequently used APIs to the newcomer. For example, there are two APIs for setting the PoolingType of a SubsamplingLayer:

- `SubsamplingLayer.Builder(PoolingType)`, where the option is set in the constructor,
- `SubsamplingBuilder.poolingType(PoolingType)`, which uses a dedicated method

In our concernified API we decided to only suggest one of the alternative API elements to the user to reduce the size of the suggested API. In our case we chose the former, since every instance of the `Layer` type needs to be instantiated using the constructor anyhow. However, the sample code `CIFARClassifier` uses the latter one.

In another case, the DL4J API offers different ways to specify activation functions. We decided to expose to the user the concrete activation enumeration `org.nd4j.linalg.`
`↪activations.Activation`, which the user can then pass as a parameter to different methods to select the activation function. We believe that this is simple, since the beginners can directly use enum constants such as `Activation.RELU`. But the sample code `CustomActivationUsageEx` demonstrates how to customize activation, and to do that it uses the interface `org.nd4j.linalg.`
`↪activations.IActivation` of the DL4J API and defines the activation function by implementing that interface.

4.5 Threats to Model Quality

Our evaluating experiment only contained 14 code samples for the complete machine learning pipeline and 7 samples for the partial pipeline. The limited number of code samples has negative impact on the experiment result. To remedy this to some extent, we made our code samples to be representative (i.e., reflecting the functionalities that beginners would most likely use) by taking some code examples from a repository of a training course for DL4J beginners to build up the experiment samples.

We also checked whether our evaluating examples cover our DL4J feature model. The coverage is shown in Figure 4.10, in which each feature of Figure 3.1 is represented by a number obtained by numbering the features that have attached API elements in the feature tree using a breadth-first traversal from left to right. That way, *Custom Data* is labelled 1 and *Customize High Level Settings* is labelled 21.

This exercise showed that while many features are well covered, 6 features out of 31 total were not used in any of the test examples. Especially the features related to the network type *ComputationGraph* were lacking examples. This makes sense, as using a *ComputationGraph* network is more complex compared to using a *MultiLayerNetwork*, and as a result the code samples for beginners rarely contain it.

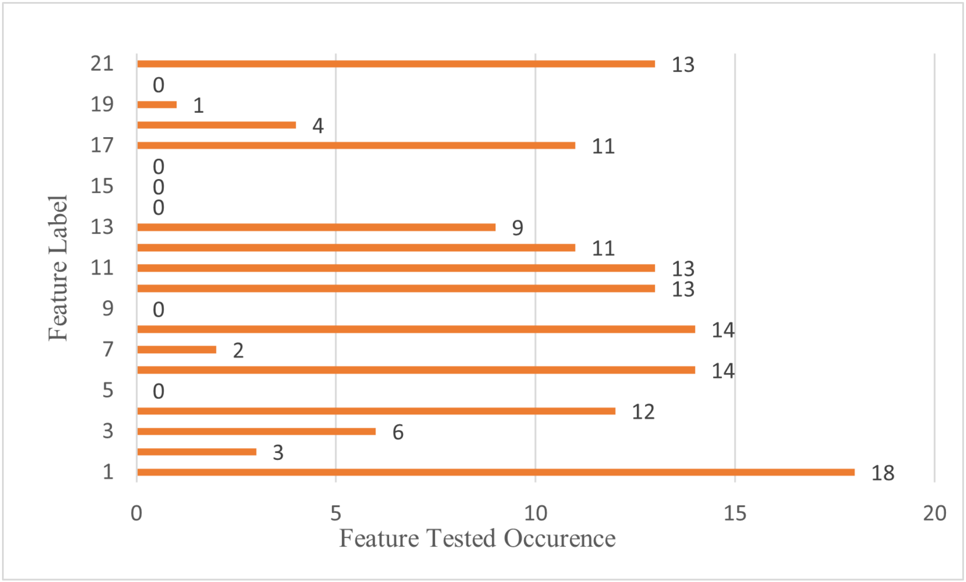


Figure 4.10: Feature Coverage in Test Code Samples

During the process of concernifying *DL4J*, even though we have tried to build the feature model in a structure that reflects the main functionality of *DL4J* organized according to the machine learning pipeline from the perspective of a concern user, we cannot guarantee the universality of our choices. We might have made subjective choices, and hence not every concern user will find our proposed feature model and feature scope intuitive. For example, as mentioned above, we categorized the APIs to convert a CSV column from `String` to `int` with the feature *Transform Data*. Some users might think it should be included in *Structure Data*, since the operation changes the data schema which defines the data structure. This difference in interpretation can lead to incomplete or wrong feature selections by a concern user, which then results in failing to generate the desired APIs.

Finally, in the process of testing, it was the concern builders who read the code samples and determined the corresponding feature selections, where again subjective factors might play a role. As concern builders we might have had some underlying knowledge of DL4J that concern users, especially beginners, will not have.

4.6 Pros and Cons

With our concernified *DL4J* API, beginners can easily get an overview of the main functionality that *DL4J* offers by reading the feature model. The information in the feature model is short and concise, compared to having to go through the lengthy informal textual documentation.

Furthermore, our approach effectively shrinks the API candidate elements that the developer needs to consider to use significantly. This reduces the complexity, and hence can result in considerable time savings and prevent confusion, especially for newcomers to ML. In our tests the API was shrunk by two orders of magnitude, i.e., from 3741 public classes and 92196 public functions to a mere 168 public classes and 531 functions.

Finally, our concernified API considers the entire ML pipeline, and deals with feature interactions transparently. As a result, the user is more likely to produce code that correctly implements a complete ML application. Furthermore, newcomers will not use incorrect combinations of API calls caused by feature selection changes at the former stages of the ML pipeline. For example, when a user switches from CSV data to image data, our feature interaction models automatically determine all preprocessing API calls that have to be updated.

The biggest drawback of our approach is unfortunately quite severe. If our concernified API incorrectly omits an API element in the realization model for a given feature selection, the concern user will not see that API element in the generated API. If there is no alternative API that achieves the desired functionality in the generated API, then the user will most likely spend an excessive amount of time looking for it, and the resulting confusion can be considerable. And in the end, the user would have to fall back to the original API.

Although this drawback is severe, we believe that such situations will occur less and less often as the approach is widely adopted, since whenever such an omission is encountered we can update the corresponding realization model. As a result, in a long run, the concernified API should be free of such omissions.

5 Conclusion and Future Work

5.1 Conclusion

Currently the documentations of the Java libraries are mostly in the form of *JavaDoc*, which defines a standard format for the automated documentation generation in HTML format from Java source code comments. Although it clearly defines the official Java API specification, since the documentation is specified at the source code level and hence modularized according to classes and methods, high-level information about the library is typically scattered across many places. For example, some of the explanations of APIs contain information that is too detailed for the developers, and they need to refer to other material to obtain a better global understanding.

Apart from the *JavaDoc*, another scattered knowledge problem faced by the beginners to learn APIs is that they also need to refer to some tutorials and read some code examples. In this scenario, the users want to dynamically interact with the documentations to get their customized recommendation of the APIs. For different application purposes, they need to read different tutorials, which makes these information of documentation scattering over too many technical blogs or forums.

Our contribution to this was an unified model enables users to dynamically interact with it to get a customized API recommendation. This one-stop service insures the information are not scattered so that beginners only need this model as their Java documentation. We used the approach of *concernification*, modeling part of the framework *DL4J* into three interfaces: variation interface, usage interface and customization interface. Concern users first make selection in the feature model of the variation interface. Based on the user's feature selection, the usage interface will provide a realization model containing an API subset of the *DL4J* framework, tailoring the API to the user's needs.

To concernify the *DL4J* framework, we first elaborated a feature model based on the tutorials and code examples. Considering the fact that machine learning has several stages from loading data to evaluating network, the features are grouped according to the deep learning pipeline with

OR or *XOR* (exclusive or) relationship among those features which have the same parent. At the same time, the non-root features have a relationship to their parents (mandatory, optional). We also added cross-tree constraints (requires or excludes) to ensure users make correct feature selection. In this way, new comers of the *DL4J* library can instantly get the understanding of what functionality it provides and make selection of features more easily.

Each feature has its corresponding realization model containing only the related API elements. We extracted parts of the realization models which might be reused by several features as a new realization model so that it can be included by different realization models sharing with this common part. Feature's behavior is not unchanging and can be influenced by the combination of feature selection, which is defined as feature interaction. For example, the behavior of feature "structure data" depends on the choice of the concrete data type in feature "load data". Generally, each feature interaction should have its corresponding conflict resolution model. However, in a feature model, there can be many options of feature selection, leading to a large combination number. To deal with the complexity of feature interactions for all possibilities of selecting features, we also extracted some common parts (if any) as general realization models. The complicated realization models which involves *K* features can be built upon those general parts corresponding to *K-1* features.

Finally, the thesis provided an experiment on *DL4J* code samples to evaluate the quality of our concernified *DL4J* library using accuracy, precision and improvement factor. Starting from the code samples, we selected those features in the feature model based on our inspection of the code. The TouchCORE tool generated a woven model (realization model result) containing the corresponding API elements, i.e., classes, methods and constructors. To compare the API elements in this model with the ones used in the code sample we wrote three utility tools: *Result Converter*, *GroundTruth Generator* and *Comparator Analyser*. The experiment achieved an average recall of 98.9% for API *classes* and 98.0% for API *methods and constructors*. The respective precision is 19.3% and 13.8%, which represents an improvement of two orders of magnitude compared to the complete *DL4J* API. Compared to the original scope of the concernified *DL4J*, which having 3741 classes (inner classes included) and 92196 public methods and constructors, such reducing of API elements significantly alleviate the users' efforts spent on searching the related APIs.

5.2 Future Work

Compared to the traditional materials like official documentation, tutorials which the beginners can refer to, a concernified library systematically presents its main functionality. Besides, users can interact with the concernified library to acquire the API usage suggestions based on their feature selection. This now opens many exciting possibilities for further research. As future work, we plan to expand the concern scope to make it applicable in wider situations. Additionally, user study can be conducted to eliminate the evaluation bias.

5.2.1 Expand Concern Scope

Because our approach aims at providing guidance to *DL4J* beginners, it contains a limited set of features for building a simple but complete ML pipeline. The current concernified APIs could be extended to include more features and as a result could apply to a broader scope. For example, besides performing learning on the local machine, *DL4J* also supports distributed neural network training, prediction and evaluation on a cluster of CPU or GPU machines using *Apache Spark*. If integrated into our concernified API, selecting the *Distributed ML* feature would give the user access to a completely different API. With the expansion of the concern scope, the complexity of feature interactions will also grow, which will need more investigation on how to abstract the models properly.

Also, currently our feature model has been built with the assumption that the users want to build an application with a complete ML pipeline. In some cases, however, users might want to write code that does not cover the entire ML workflow. For example, one can envision code that simply loads data and pre-processes it, but then stores the processed data so it can be used at a later point in time. To support such use cases, our feature model would have to be adapted, and in particular some of the realization models would have to be updated.

Additionally, the current concern is built for beginner users of DL4J, and hence its feature model presents a coarse-grained, high-level view of the framework. We can adapt the approach of *framework-specific models* [2] introduced in Section 2.2.3 to construct another concern of DL4J specifically targetting experienced developers, enabling more flexibility in customization and usage of the concern.

5.2.2 Improve the Experiment

The precision rate can not fully reflect model ability in retrieving relevant information because some of the API elements which are not belong to the ground truth have certain relationship with the ground truth API elements. In the future, we can calculate correlation coefficient between API elements in $S_{Gen} - S_{GroundTruth}$ and $S_{GroundTruth}$ to further analyze the model information retrieving ability.

5.2.3 Conduct User Study

While our initial evaluation suggests that our concernified API would benefit developers and in particular new comers, it would be interesting to conduct an actual user study to confirm this. Users can provide their opinions about the concern, especially its usability. For instance, if some wording of the feature model hinder the users from understanding because of ambiguity.

Furthermore, for the experiments in Section 4.2, the features were selected by the concern builders, assuming the concern users will also make such feature selections. Having beginner users of *DL4J* actually take part in the experiments would reveal whether they too would make those selections, making the evaluation more convincing.

Bibliography

[1] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Engin Yildirim, Omar Alam, and Jörg Kienzle. Touchram: A multitouch-enabled tool for aspect-oriented software design. In *International Conference on Software Language Engineering*, pages 275–285. Springer, 2012. 2.1.2.1, 2.1.4

[2] Michał Antkiewicz, Krzysztof Czarnecki, and Matthew Stephan. Engineering of framework-specific modeling languages. *IEEE Transactions on Software Engineering*, 35(6):795–824, 2009. 2.2.3, 5.2.1

[3] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *International conference on software product lines*, pages 266–283. Springer, 2004. 2.2.3

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. 3.3.1

[5] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 75–88, 2006. 2.2.3

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. 2.2.2

[7] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936. 3.3.1

[8] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 631–642, 2016. 2.2.2

[9] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. In *2018 33rd IEEE/ACM*

International Conference on Automated Software Engineering (ASE), pages 293–304. IEEE, 2018. 2.2.2

[10] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990. 2.1.1

[11] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. Vcu: the three dimensions of reuse. In *International Conference on Software Reuse*, pages 122–137. Springer, 2016. 2.1.2.2, 2.1.4

[12] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). 3.3.1

[13] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. 3.3.1

[14] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009. 2.2.2

[15] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. Generating query-specific class api summaries. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 120–130, 2019. 2.2.2

[16] James Martin and Jin LC Guo. Deep api learning revisited. *arXiv preprint arXiv:2205.01254*, 2022. 2.2.2

[17] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Alvaro Lopez Garcia, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52(1):77–124, 2019. 3.1

[18] Elisabeth Niehaus, Klaus Pohl, and Günter Böckle. Software product line engineering: Foundations, principles and techniques, kapitel product management, 2005. 2.1.1

[19] Josh Patterson and Adam Gibson. *Deep learning: A practitioner's approach.* " O'Reilly Media, Inc.", 2017. 3.2, 3.3.1

[20] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Leveraging crowd knowledge for software comprehension and development. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 57–66. IEEE, 2013. 2.2.2

[21] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 357–367. IEEE, 2016. 2.2.2

[22] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009. 2.2.4

[23] Matthias Schöttle and Jörg Kienzle. Concern-oriented interfaces for model-based reuse of apis. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 286–291. IEEE, 2015. 2.1.3, 2.1.3

[24] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th international conference on software engineering*, pages 643–652, 2014. 2.2.1

[25] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119, 1999. 2.1.2

[26] Christoph Treude and Martin P Robillard. Augmenting api documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 392–403. IEEE, 2016. 2.2.1

[27] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining api usage scenarios from stack overflow. *Information and Software Technology*, 122:106277, 2020. 2.2.1

[28] Gias Uddin and Martin P Robillard. How api documentation fails. *Ieee software*, 32(4):68–75, 2015. 1.1, 2.2.2

[29] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage api usage patterns from source code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 319–328. IEEE, 2013. 2.2.2

[30] Hang Yin, Yuanhao Zheng, Yanchun Sun, and Gang Huang. An api learning service for inexperienced developers based on api knowledge graph. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 251–261. IEEE, 2021. 2.2.2

[31] Shichao Zhang, Chengqi Zhang, and Qiang Yang. Data preparation for data mining. *Applied artificial intelligence*, 17(5-6):375–381, 2003. 3.3.1

A

Code Sample in Section 3.6

```
import org.deeplearning4j.datasets.iterator.impl.MnistDataSetIterator;
import org.deeplearning4j.nn.conf.MultiLayerConfiguration;
import org.deeplearning4j.nn.conf.NeuralNetConfiguration;
import org.deeplearning4j.nn.conf.layers.DenseLayer;
import org.deeplearning4j.nn.conf.layers.OutputLayer;
import org.deeplearning4j.nn.multilayer.MultiLayerNetwork;
import org.deeplearning4j.nn.weights.WeightInit;
import org.deeplearning4j.optimize.listeners.ScoreIterationListener;
import org.nd4j.evaluation.classification.Evaluation;
import org.nd4j.linalg.activations.Activation;
import org.nd4j.linalg.dataset.api.iterator.DataSetIterator;
import org.nd4j.linalg.learning.config.Nadam;
import org.nd4j.linalg.lossfunctions.LossFunctions.LossFunction;

/** A slightly more involved multilayered (MLP) applied to digit
   classification for the MNIST dataset
   (http://yann.lecun.com/exdb/mnist/).
* This example uses two input layers and one hidden layer.
*
* The first input layer has input dimension of numRows*numColumns
   where these variables indicate the
* number of vertical and horizontal pixels in the image. This layer
   uses a rectified linear unit
* (relu) activation function. The weights for this layer are
   initialized by using Xavier initialization
```

```
* to avoid having a steep learning curve. This layer sends 500 output
   signals to the second layer.
*
* The second input layer has input dimension of 500. This layer also
   uses a rectified linear unit
* (relu) activation function. The weights for this layer are also
   initialized by using Xavier initialization to avoid having a steep
   learning curve. This layer sends 100 output signals to the hidden
   layer.
*
* The hidden layer has input dimensions of 100. These are fed from the
   second input layer. The weights
* for this layer is also initialized using Xavier initialization. The
   activation function for this
* layer is a softmax, which normalizes all the 10 outputs such that
   the normalized sums
* add up to 1. The highest of these normalized values is picked as the
   predicted class.
*/
public class MNISTDoubleLayer {

   public static void main(String[] args) throws Exception {
      //number of rows and columns in the input pictures
      final int numRows = 28;
      final int numColumns = 28;
      int outputNum = 10; // number of output classes
      int batchSize = 64; // batch size for each epoch
      int rngSeed = 123; // random number seed for reproducibility
      int numEpochs = 15; // number of epochs to perform
      double rate = 0.0015; // learning rate

      //Get the DataSetIterators:
      DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,
         true, rngSeed);
      DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,
         false, rngSeed);
```

Code Sample in Section 3.6

```
      MultiLayerConfiguration conf = new
         NeuralNetConfiguration.Builder()
         .seed(rngSeed) //include a random seed for reproducibility
         .activation(Activation.RELU)
         .weightInit(WeightInit.XAVIER)
         .updater(new Nadam())
         .l2(rate * 0.005) // regularize learning model
         .list()
         .layer(new DenseLayer.Builder() //create the first input
            layer.
               .nIn(numRows * numColumns)
               .nOut(500)
               .build())
         .layer(new DenseLayer.Builder() //create the second input
            layer
               .nIn(500)
               .nOut(100)
               .build())
         .layer(new
            OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
            //create hidden layer
               .activation(Activation.SOFTMAX)
               .nOut(outputNum)
               .build())
         .build();

      MultiLayerNetwork model = new MultiLayerNetwork(conf);
      model.init();
      model.setListeners(new ScoreIterationListener(5));

      model.fit(mnistTrain, numEpochs);

      Evaluation eval = model.evaluate(mnistTest);
   }
}
```

B

Woven Model Result for the Code Sample

The classes, APIs, constructors generated in the woven model for the code sample in section 3.6 are listed here. We use brown color to point out those are used in the code sample.

```
------CLASSES------
org.deeplearning4j.nn.conf.layers.CenterLossOutputLayer
org.deeplearning4j.nn.conf.layers.ConvolutionLayer
org.deeplearning4j.datasets.iterator.impl.IrisDataSetIterator
org.nd4j.linalg.dataset.api.iterator.DataSetIterator
org.deeplearning4j.optimize.listeners.EvaluativeListener
org.deeplearning4j.nn.conf.layers.OutputLayer
org.deeplearning4j.datasets.iterator.impl.MnistDataSetIterator
org.deeplearning4j.nn.conf.layers.DenseLayer
java.lang.Integer
org.nd4j.linalg.learning.config.Sgd
org.deeplearning4j.nn.conf.layers.BatchNormalization
org.deeplearning4j.nn.conf.layers.GlobalPoolingLayer
org.nd4j.linalg.schedule.ISchedule
org.deeplearning4j.nn.layers.BaseLayer<>
org.deeplearning4j.optimize.api.IterationListener
org.deeplearning4j.nn.conf.NeuralNetConfiguration.ListBuilder
org.deeplearning4j.nn.weights.WeightInit
org.nd4j.linalg.learning.AdamUpdater
org.nd4j.linalg.learning.config.AdaGrad
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder
org.nd4j.linalg.dataset.DataSet
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.PoolingType
```

Woven Model Result for the Code Sample

```
org.deeplearning4j.optimize.listeners.ScoreIterationListener
org.nd4j.linalg.factory.Nd4j
org.deeplearning4j.nn.conf.layers.GlobalPoolingLayer.Builder
org.nd4j.linalg.learning.AdaDeltaUpdater
org.nd4j.linalg.learning.config.AMSGrad
org.deeplearning4j.optimize.api.TrainingListener
org.nd4j.linalg.lossfunctions.impl.LossMCXENT
org.nd4j.evaluation.classification.Evaluation
org.deeplearning4j.nn.conf.layers.Layer
org.nd4j.linalg.learning.RmsPropUpdater
org.nd4j.linalg.learning.AMSGradUpdater
java.util.Map<Integer,Double>
org.deeplearning4j.nn.api.OptimizationAlgorithm
org.deeplearning4j.datasets.iterator.impl.TinyImageNetDataSetIterator
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.Builder
org.deeplearning4j.nn.conf.layers.FeedForwardLayer.Builder
org.deeplearning4j.optimize.api.BaseTrainingListener
org.nd4j.linalg.api.rng.Random
org.nd4j.linalg.learning.config.Nadam
java.lang.Double
org.nd4j.linalg.dataset.api.DataSet
org.nd4j.linalg.learning.NoOpUpdater
org.deeplearning4j.nn.conf.layers.BaseLayer.Builder
org.deeplearning4j.nn.conf.layers.SubsamplingLayer
org.nd4j.linalg.learning.config.IUpdater
org.nd4j.linalg.learning.config.AdaDelta
org.nd4j.linalg.schedule.StepSchedule
org.deeplearning4j.datasets.iterator.impl.Cifar10DataSetIterator
org.deeplearning4j.nn.conf.layers.FeedForwardLayer
org.nd4j.evaluation.regression.RegressionEvaluation
org.deeplearning4j.nn.conf.layers.BatchNormalization.Builder
org.nd4j.linalg.schedule.MapSchedule
org.nd4j.linalg.lossfunctions.LossFunctions.LossFunction
org.nd4j.linalg.lossfunctions.ILossFunction
org.nd4j.linalg.dataset.SplitTestAndTrain
org.nd4j.linalg.learning.config.RmsProp
```

Woven Model Result for the Code Sample

```
org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder
org.nd4j.linalg.activations.Activation
org.deeplearning4j.nn.conf.layers.CenterLossOutputLayer.Builder
org.nd4j.linalg.learning.NesterovsUpdater
org.nd4j.linalg.learning.NadamUpdater
org.deeplearning4j.nn.conf.layers.OutputLayer.Builder
org.nd4j.linalg.learning.config.NoOp
org.datavec.image.transform.ImageTransform
org.nd4j.linalg.learning.SgdUpdater
org.deeplearning4j.nn.conf.NeuralNetConfiguration
org.deeplearning4j.nn.multilayer.MultiLayerNetwork
org.deeplearning4j.nn.layers.BaseOutputLayer<>
org.deeplearning4j.nn.conf.inputs.InputType
org.deeplearning4j.nn.conf.MultiLayerConfiguration
org.nd4j.linalg.api.ndarray.INDArray
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.Builder
org.deeplearning4j.nn.conf.layers.DropoutLayer
org.nd4j.linalg.learning.config.Nesterovs
org.nd4j.linalg.learning.config.Adam
org.deeplearning4j.nn.conf.layers.BaseOutputLayer.Builder
org.nd4j.linalg.learning.AdaGradUpdater
------END OF CLASSES------
------API CALLS------
org.deeplearning4j.nn.conf.layers.
   ↪CenterLossOutputLayer.Builder.gradientCheck(boolean)
org.deeplearning4j.nn.conf.layers.
   ↪CenterLossOutputLayer.Builder.lambda(double)
org.deeplearning4j.nn.multilayer.
   ↪MultiLayerNetwork.fit(org.nd4j.linalg.dataset.api.DataSet)
org.deeplearning4j.nn.conf.layers.OutputLayer.Builder.build()
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪output(org.nd4j.linalg.dataset.api.iterator.DataSetIterator)
org.deeplearning4j.nn.conf.layers.BaseLayer.Builder.
   ↪updater(org.nd4j.linalg.learning.config.IUpdater)
org.deeplearning4j.nn.conf.layers.FeedForwardLayer.Builder.nOut(long)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder.
   ↪weightInit(org.deeplearning4j.nn.weights.WeightInit)
org.nd4j.linalg.dataset.DataSet.sample(int,
   org.nd4j.linalg.api.rng.Random)
```

Woven Model Result for the Code Sample

```
org.deeplearning4j.nn.conf.layers.FeedForwardLayer.Builder.nIn(long)
org.nd4j.evaluation.classification.Evaluation.accuracy()
org.nd4j.evaluation.regression.RegressionEvaluation.stats()
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪fit(org.nd4j.linalg.dataset.api.iterator.DataSetIterator)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪output(org.nd4j.linalg.dataset.api.iterator.DataSetIterator,
   boolean)
org.nd4j.linalg.dataset.DataSet.splitTestAndTrain(int)
org.deeplearning4j.nn.conf.layers.BaseLayer.Builder.l1(double)
org.nd4j.linalg.factory.Nd4j.create(double[][])
org.nd4j.linalg.factory.Nd4j.create(double[][][])
org.nd4j.linalg.factory.Nd4j.create(int[][])
org.deeplearning4j.nn.conf.inputs.InputType.convolutional(long,
   long, long)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪output(org.nd4j.linalg.api.ndarray.INDArray)
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.Builder.nIn(int)
org.nd4j.linalg.dataset.api.iterator.DataSetIterator.hasNext()
org.deeplearning4j.nn.conf.layers.GlobalPoolingLayer.Builder.
   ↪poolingType(org.deeplearning4j.nn.conf.layers.PoolingType)
org.deeplearning4j.nn.conf.inputs.InputType.convolutionalFlat(long,
   long, long)
org.deeplearning4j.datasets.iterator.impl.
   ↪Cifar10DataSetIterator.next(int)
org.deeplearning4j.nn.conf.layers.BaseLayer.Builder.
   ↪activation(org.nd4j.linalg.activations.Activation)
org.deeplearning4j.nn.conf.layers.BaseOutputLayer.Builder.
   ↪lossFunction(org.nd4j.linalg.
   ↪lossfunctions.LossFunctions.LossFunction)
org.nd4j.linalg.factory.Nd4j.create(int[])
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder.nInt(int)
org.deeplearning4j.nn.conf.layers.GlobalPoolingLayer.Builder.
   ↪poolingDimensions(int...)
org.nd4j.linalg.factory.Nd4j.vstack(int...)
org.deeplearning4j.nn.conf.layers.
   ↪ConvolutionLayer.Builder.stride(int...)
org.nd4j.linalg.factory.Nd4j.create(double[])
org.deeplearning4j.nn.conf.inputs.InputType.convolutional3D(int,
   int, int, int)
```

Woven Model Result for the Code Sample

```
org.nd4j.linalg.dataset.DataSet.numInputs()
org.deeplearning4j.nn.conf.layers.
   ↪ConvolutionLayer.Builder.padding(int...)
org.deeplearning4j.nn.conf.layers.
   ↪SubsamplingLayer.Builder.stride(int...)
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.Builder.nOut(int)
org.deeplearning4j.datasets.iterator.impl.
   ↪IrisDataSetIterator.hasNext()
org.deeplearning4j.datasets.iterator.impl.
   ↪TinyImageNetDataSetIterator.hasNext()
org.nd4j.linalg.dataset.DataSet.splitTestAndTrain(double)
org.deeplearning4j.nn.conf.layers.
   ↪SubsamplingLayer.Builder.padding(int...)
org.nd4j.linalg.dataset.SplitTestAndTrain.getTest()
org.nd4j.linalg.dataset.api.iterator.DataSetIterator.next(int)
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.Builder.build()
org.deeplearning4j.datasets.iterator.impl.
   ↪TinyImageNetDataSetIterator.next(int)
org.deeplearning4j.nn.conf.layers.OutputLayer.Builder.dropOut(double)
org.deeplearning4j.nn.conf.layers.BaseLayer.Builder.l2(double)
org.nd4j.linalg.dataset.api.iterator.DataSetIterator.reset()
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪evaluateRegression(org.nd4j.linalg.dataset.api.
   ↪iterator.DataSetIterator)
org.nd4j.linalg.factory.Nd4j.create(float[])
org.deeplearning4j.nn.conf.inputs.InputType.feedForward(long)
org.deeplearning4j.nn.multilayer.
   ↪MultiLayerNetwork.fit(org.nd4j.linalg.api.ndarray.INDArray,
   org.nd4j.linalg.api.ndarray.INDArray)
org.nd4j.linalg.factory.Nd4j.create(float[][])
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.
   ↪Builder.kernelSize(int...)
org.nd4j.linalg.factory.Nd4j.hstack(int...)
org.nd4j.linalg.dataset.DataSet.numExamples()
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪Builder.activation(org.nd4j.linalg.activations.Activation)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪Builder.updater(org.nd4j.linalg.learning.config.IUpdater)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.init()
org.nd4j.evaluation.classification.Evaluation.eval(int, int)
org.nd4j.linalg.factory.Nd4j.create(int[][][])
```

Woven Model Result for the Code Sample

```
org.deeplearning4j.nn.conf.layers.OutputLayer.Builder.nOut(int)
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder.build()
org.nd4j.linalg.dataset.DataSet.getLabels()
org.deeplearning4j.nn.conf.layers.CenterLossOutputLayer.
   ↪Builder.alpha(double)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪output(org.nd4j.linalg.api.ndarray.INDArray, boolean)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.summary()
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪setListeners(org.deeplearning4j.optimize.api.TrainingListener...)
org.deeplearning4j.nn.conf.layers.OutputLayer.Builder.
   ↪activation(org.nd4j.linalg.activations.Activation)
org.deeplearning4j.nn.conf.inputs.InputType.feedForward(int)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪evaluate(org.nd4j.linalg.dataset.api.iterator.DataSetIterator)
org.nd4j.evaluation.classification.Evaluation.stats()
org.deeplearning4j.nn.conf.inputs.InputType.convolutional(int, int,
   int)
org.deeplearning4j.datasets.iterator.impl.
   ↪MnistDataSetIterator.next(int)
org.nd4j.linalg.dataset.DataSet.sample(int, boolean)
org.deeplearning4j.nn.conf.layers.GlobalPoolingLayer.Builder.build()
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪fit(org.nd4j.linalg.dataset.api.iterator.DataSetIterator, int)
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.Builder.build()
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.
   ↪Builder.activation(int)
org.deeplearning4j.nn.conf.inputs.InputType.
   ↪convolutionalFlat(int, int, int)
org.nd4j.linalg.dataset.SplitTestAndTrain.getTrain()
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.
   ↪Builder.kernelSize(int...)
org.nd4j.evaluation.classification.Evaluation.
   ↪eval(org.nd4j.linalg.api.ndarray.INDArray,
   org.nd4j.linalg.api.ndarray.INDArray)
org.deeplearning4j.nn.conf.layers.FeedForwardLayer.Builder.nIn(int)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.ListBuilder.build()
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪ListBuilder.layer(org.deeplearning4j.nn.conf.layers.Layer)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder.
   ↪optimizationAlgo(org.deeplearning4j.nn.api.OptimizationAlgorithm)
org.nd4j.evaluation.classification.Evaluation.confusionToString()
```

Woven Model Result for the Code Sample

```
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪ListBuilder.setInputType(int)
org.deeplearning4j.datasets.iterator.impl.IrisDataSetIterator.next(int)
org.deeplearning4j.nn.conf.layers.FeedForwardLayer.Builder.nOut(int)
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder.nIn(long)
org.nd4j.linalg.dataset.DataSet.numOutcomes()
org.deeplearning4j.datasets.iterator.impl.
   ↪TinyImageNetDataSetIterator.next()
org.deeplearning4j.nn.conf.layers.BatchNormalization.Builder.build()
org.deeplearning4j.datasets.iterator.impl.IrisDataSetIterator.next()
org.nd4j.linalg.dataset.DataSet.getFeatures()
org.nd4j.linalg.factory.Nd4j.create(float[][][])
org.deeplearning4j.datasets.iterator.impl.Cifar10DataSetIterator.next()
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪Builder.learningRate(double)
org.deeplearning4j.nn.conf.inputs.InputType.convolutional3D(long,
   long, long, long)
org.deeplearning4j.datasets.iterator.impl.
   ↪Cifar10DataSetIterator.hasNext()
org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder.l2(double)
org.deeplearning4j.nn.conf.layers.BaseLayer.Builder.
   ↪weightInit(org.deeplearning4j.nn.weights.WeightInit)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder.list()
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.
   ↪doEvaluation(org.nd4j.linalg.dataset.api.iterator.DataSetIterator,
   T...)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.ListBuilder.
   ↪setInputType(org.deeplearning4j.nn.conf.inputs.InputType)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪Builder.miniBatch(boolean)
org.nd4j.evaluation.classification.Evaluation.
   ↪stats(org.nd4j.linalg.api.ndarray.INDArray,
   org.nd4j.linalg.api.ndarray.INDArray)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪.biasInit(double)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.fit()
org.deeplearning4j.nn.conf.layers.OutputLayer.
   ↪Builder.name(java.lang.String)
org.nd4j.linalg.dataset.DataSet.sample(int)
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder.nOut(long)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
```

```
   ↪ListBuilder.layer(int, org.deeplearning4j.nn.conf.layers.Layer)
org.deeplearning4j.datasets.iterator.impl.
   ↪MnistDataSetIterator.hasNext()
org.nd4j.linalg.dataset.DataSet.get(int)
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder.nOut(int)
org.nd4j.evaluation.classification.Evaluation.confusionMatrix()
org.deeplearning4j.datasets.iterator.impl.
   ↪MnistDataSetIterator.next()
org.deeplearning4j.nn.conf.layers.
   ↪CenterLossOutputLayer.Builder.build()
org.deeplearning4j.nn.conf.NeuralNetConfiguration.Builder.seed(long)
------END OF API CALLS------
------CONSTRUCTORS------
org.deeplearning4j.datasets.iterator.impl.
   ↪IrisDataSetIterator.IrisDataSetIterator()
org.nd4j.linalg.learning.config.Nadam.Nadam(double)
org.deeplearning4j.nn.conf.layers.CenterLossOutputLayer.
   ↪Builder.CenterLossOutputLayer$Builder()
org.nd4j.linalg.learning.config.Adam.Adam(double, double, double,
   double)
org.deeplearning4j.datasets.iterator.impl.Cifar10DataSetIterator.
   ↪Cifar10DataSetIterator(int, int[],
   org.deeplearning4j.datasets.fetchers.DataSetType)
org.deeplearning4j.nn.conf.layers.OutputLayer.
   ↪Builder.OutputLayer$Builder()
org.nd4j.linalg.learning.config.AMSGrad.AMSGrad(double)
org.deeplearning4j.optimize.api.IterationListener.IterationListener()
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.
   ↪Builder.SubsamplingLayer$Builder(int...)
org.deeplearning4j.datasets.iterator.impl.Cifar10DataSetIterator.
   ↪Cifar10DataSetIterator(int, int[],
   org.deeplearning4j.datasets.fetchers.DataSetType,
   org.datavec.image.transform.ImageTransform, long)
org.nd4j.linalg.learning.config.Adam.Adam()
org.nd4j.linalg.learning.config.RmsProp.RmsProp()
org.nd4j.linalg.learning.NadamUpdater.
   ↪NadamUpdater(org.nd4j.linalg.learning.config.Nadam)
org.nd4j.linalg.dataset.SplitTestAndTrain.
   ↪SplitTestAndTrain(org.nd4j.linalg.dataset.DataSet,
   org.nd4j.linalg.dataset.DataSet)
org.deeplearning4j.nn.conf.layers.BatchNormalization.BatchNormalization()
```

Woven Model Result for the Code Sample

```
org.deeplearning4j.nn.conf.layers.CenterLossOutputLayer.
   ↪Builder.CenterLossOutputLayer$Builder(org.nd4j.linalg.
   ↪lossfunctions.LossFunctions.LossFunction)
org.nd4j.linalg.learning.config.AdaGrad.AdaGrad()
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.
   ↪Builder.SubsamplingLayer$Builder()
org.deeplearning4j.nn.conf.layers.BaseOutputLayer.
   ↪Builder.BaseOutputLayer$Builder(org.nd4j.
   ↪linalg.lossfunctions.ILossFunction)
org.deeplearning4j.nn.conf.layers.GlobalPoolingLayer.
   ↪Builder.GlobalPoolingLayer$Builder()
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.
   ↪Builder.ConvolutionLayer$Builder(int...)
org.deeplearning4j.datasets.iterator.impl.Cifar10DataSetIterator.
   ↪Cifar10DataSetIterator(int,
   org.deeplearning4j.datasets.fetchers.DataSetType)
org.deeplearning4j.nn.conf.layers.BaseOutputLayer.
   ↪Builder.BaseOutputLayer$Builder()
org.deeplearning4j.nn.conf.layers.OutputLayer.
   ↪Builder.OutputLayer$Builder(org.nd4j.linalg.
   ↪lossfunctions.ILossFunction)
org.nd4j.evaluation.regression.RegressionEvaluation.
   ↪RegressionEvaluation(java.lang.String...)
org.deeplearning4j.nn.conf.layers.ConvolutionLayer.
   ↪Builder.ConvolutionLayer$Builder()
org.deeplearning4j.nn.conf.layers.FeedForwardLayer.
   ↪Builder.FeedForwardLayer$Builder()
org.nd4j.linalg.learning.SgdUpdater.
   ↪SgdUpdater(org.nd4j.linalg.learning.config.Sgd)
org.deeplearning4j.datasets.iterator.impl.
   ↪TinyImageNetDataSetIterator.TinyImageNetDataSetIterator(int,
   int[], org.deeplearning4j.datasets.fetchers.DataSetType,
   org.datavec.image.transform.ImageTransform, long)
org.deeplearning4j.datasets.iterator.impl.
   ↪TinyImageNetDataSetIterator.TinyImageNetDataSetIterator(int)
org.nd4j.evaluation.regression.
   ↪RegressionEvaluation.RegressionEvaluation()
org.deeplearning4j.datasets.iterator.impl.
   ↪MnistDataSetIterator.MnistDataSetIterator(int, int)
org.nd4j.evaluation.regression.
   ↪RegressionEvaluation.RegressionEvaluation(long)
org.deeplearning4j.nn.conf.NeuralNetConfiguration.
   ↪Builder.NeuralNetConfiguration$Builder()
org.nd4j.linalg.learning.AMSGradUpdater.
   ↪AMSGradUpdater(org.nd4j.linalg.learning.config.AMSGrad)
```

Woven Model Result for the Code Sample

```
org.nd4j.linalg.learning.config.
   ↪Adam.Adam(org.nd4j.linalg.schedule.ISchedule)
org.deeplearning4j.datasets.iterator.impl.
   ↪Cifar10DataSetIterator.Cifar10DataSetIterator(int)
org.deeplearning4j.datasets.iterator.impl.
   ↪MnistDataSetIterator.MnistDataSetIterator(int, boolean, int)
org.nd4j.linalg.learning.config.Nadam.Nadam(double, double, double,
   double)
org.nd4j.linalg.lossfunctions.impl.LossMCXENT.
   ↪LossMCXENT(double, org.nd4j.linalg.api.ndarray.INDArray)
org.nd4j.linalg.learning.config.AdaDelta.AdaDelta()
org.nd4j.linalg.learning.NoOpUpdater.
   ↪NoOpUpdater(org.nd4j.linalg.learning.config.NoOp)
org.deeplearning4j.nn.conf.layers.BatchNormalization.
   ↪Builder.BatchNormalization$Builder()
org.deeplearning4j.datasets.iterator.impl.TinyImageNetDataSetIterator.
   ↪TinyImageNetDataSetIterator(int,
   org.deeplearning4j.datasets.fetchers.DataSetType)
org.deeplearning4j.nn.multilayer.MultiLayerNetwork.MultiLayerNetwork
   ↪(org.deeplearning4j.nn.conf.MultiLayerConfiguration)
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.
   ↪Builder.SubsamplingLayer$Builder(org.deeplearning4j.nn.
   ↪conf.layers.PoolingType)
org.nd4j.linalg.learning.config.Nadam.Nadam()
org.deeplearning4j.optimize.listeners.
   ↪EvaluativeListener.EvaluativeListener(org.nd4j.linalg.dataset.api.
   ↪iterator.DataSetIterator, int,
   org.deeplearning4j.optimize.api.InvocationType)
org.nd4j.linalg.lossfunctions.impl.LossMCXENT.LossMCXENT(org.nd4j.linalg.
   ↪api.ndarray.INDArray)
org.nd4j.linalg.learning.config.Sgd.Sgd(double)
org.nd4j.linalg.learning.config.AMSGrad.AMSGrad(double, double,
   double, double)
org.deeplearning4j.nn.conf.layers.DenseLayer.Builder.DenseLayer$Builder()
org.nd4j.linalg.learning.config.Nesterovs.Nesterovs(double, double)
org.deeplearning4j.optimize.listeners.
   ↪ScoreIterationListener.ScoreIterationListener()
org.nd4j.linalg.learning.AdaDeltaUpdater.
   ↪AdaDeltaUpdater(org.nd4j.linalg.learning.config.AdaDelta)
org.nd4j.linalg.learning.config.AdaDelta.AdaDelta(double, double)
org.nd4j.linalg.learning.config.Adam.Adam(double)
org.deeplearning4j.optimize.listeners.
   ↪EvaluativeListener.EvaluativeListener(org.nd4j.linalg.dataset.api.
    ↪iterator.DataSetIterator, int)
org.nd4j.evaluation.classification.Evaluation.Evaluation()
org.deeplearning4j.datasets.iterator.impl.
    ↪IrisDataSetIterator.IrisDataSetIterator(int, int)
org.nd4j.linalg.learning.config.Nesterovs.Nesterovs()
org.nd4j.linalg.lossfunctions.impl.LossMCXENT.LossMCXENT()
org.deeplearning4j.nn.conf.layers.DropoutLayer.DropoutLayer(double)
org.deeplearning4j.optimize.api.
    ↪BaseTrainingListener.BaseTrainingListener()
org.deeplearning4j.nn.conf.layers.SubsamplingLayer.
    ↪Builder.SubsamplingLayer$Builder(org.deeplearning4j.nn.conf.
    ↪layers.SubsamplingLayer.PoolingType)
org.nd4j.linalg.schedule.StepSchedule.StepSchedule
    ↪(org.nd4j.linalg.schedule.ScheduleType, double, double, double)
org.nd4j.linalg.schedule.MapSchedule.MapSchedule(org.nd4j.linalg.
    ↪schedule.ScheduleType, java.util.Map)
org.nd4j.linalg.learning.config.RmsProp.RmsProp(double)
org.nd4j.linalg.learning.config.Nesterovs.Nesterovs(double)
org.deeplearning4j.optimize.listeners.
    ↪ScoreIterationListener.ScoreIterationListener(int)
org.nd4j.linalg.learning.config.AdaGrad.AdaGrad(double, double)
org.nd4j.linalg.learning.RmsPropUpdater.
    ↪RmsPropUpdater(org.nd4j.linalg.learning.config.RmsProp)
org.deeplearning4j.datasets.iterator.impl.
    ↪TinyImageNetDataSetIterator.TinyImageNetDataSetIterator(int,
    int[], org.deeplearning4j.datasets.fetchers.DataSetType)
org.nd4j.linalg.learning.config.Sgd.Sgd()
org.nd4j.evaluation.classification.Evaluation.Evaluation(int)
org.nd4j.linalg.learning.config.AMSGrad.AMSGrad()
org.deeplearning4j.nn.conf.layers.OutputLayer.
    ↪Builder.OutputLayer$Builder(org.nd4j.linalg.
    ↪lossfunctions.LossFunctions.LossFunction)
org.nd4j.linalg.learning.config.Nesterovs.
    ↪Nesterovs(org.nd4j.linalg.schedule.ISchedule)
org.nd4j.linalg.learning.config.NoOp.NoOp()
org.nd4j.linalg.learning.config.AdaGrad.AdaGrad(double)
org.nd4j.linalg.learning.config.RmsProp.
    ↪RmsProp(double, double, double)
org.nd4j.linalg.learning.NesterovsUpdater.
    ↪NesterovsUpdater(org.nd4j.linalg.learning.config.Nesterovs)
org.nd4j.linalg.learning.AdaGradUpdater.
    ↪AdaGradUpdater(org.nd4j.linalg.learning.config.AdaGrad)
org.nd4j.linalg.learning.config.Nesterovs.
```

```
   ↪Nesterovs(double, org.nd4j.linalg.schedule.ISchedule)
org.nd4j.linalg.learning.AdamUpdater.
   ↪AdamUpdater(org.nd4j.linalg.learning.config.Adam)

------END OF CONSTRUCTORS------
```