

Automated Construction of Data Science Workflows

Steven Pan

A thesis presented for the degree of
Master of Computational Science



Institute of Computer Science
University of Potsdam
Germany
October 1, 2023

Abstract

In an era where data plays a key role in nearly all fields, the ability to compose workflows has never been more crucial. Automated solutions could not only speed up data analysis but also make it more accessible to non-data scientists or experts in other domains. The Automated Pipeline Explorer (APE) has successfully synthesized scientific workflows in various other domains, providing semantic abstraction to the technical tool layer. This thesis aims to answer the research question of whether APE can be applied to generate executable workflows in the data science field. To this end, it models a new domain ontology, integrates it into the APE synthesis process, and evaluates the resulting system. Furthermore, it also presents an alternative solving backend based on Answer Set Programming (ASP), extending the possibilities of APE, and peeks into the potential options of using generative artificial intelligence (AI) in workflow construction.

Zusammenfassung In einer Zeit, wo Daten eine primäre Rolle in nahezu allen Bereichen spielen, ist die Fähigkeit Workflows zu erstellen noch nie wichtiger gewesen. Automatisierte Lösungen könnten nicht nur die Datenanalyse beschleunigen, sondern diese auch zugänglicher zu Nicht-Data-Scientisten oder Experten anderer Bereiche machen. Das Tool Automated Pipeline Explorer (APE) hat bereits in einigen Domänen erfolgreich wissenschaftliche Datenverarbeitungs-Workflows erstellt und dabei eine semantische Abstraktionsebene zu den technischen Details der Tools geschaffen. Diese Arbeit versucht, die Forschungsfrage zu beantworten, ob APE auch für die Generierung von ausführbaren Workflows im Bereich Data Science verwendet werden kann. Dazu wird eine neue Domänenontologie erstellt, diese in den APE Syntheseprozess integriert und das resultierende System evaluiert. Außerdem wird ein alternativer Lösungsansatz basierend auf Answer Set Programming (ASP) vorgestellt, der die Möglichkeiten von APE erweitert, und einen Blick auf die potentiellen Ansätze der Verwendung von generativer Künstlicher Intelligenz (KI) in der Workflow-Konstruktion geworfen.

Declaration of Originality

I hereby declare that this thesis is the product of my own work. All the assistance received in preparing this thesis and the sources used have been acknowledged.

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die von mir angegebenen Hilfsmittel genutzt habe.

October 1, 2023

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In an era where data plays a key role in nearly all fields, the ability to compose workflows has never been more crucial. Automated solutions could not only speed up data analysis but also make it more accessible to non-data scientists or experts in other domains. However, the large number of tools and their complex interactions pose unique challenges to the effectiveness of automated workflow construction systems. The tool **ape!** (**ape!**)[kasalica2022synthesis] has successfully synthesized scientific workflows in various domains, such as geovisualization[kasalica2019workflow] and bioinformatics[kasalica2021ape], with promising results. The synthesized workflows use types and tools from domain-specific ontologies not suitable for general-purpose data science tasks.

This thesis aims to answer the research question of whether **ape!** can be applied to generate executable workflows in the data science field. A success could enable less skilled data scientists to effectively receive drafts of required tool sequences and experts to more efficiently explore variants of their workflows using tools from the vast number of available libraries, all while interacting with the system on a semantic abstraction layer. To this end, the thesis needs to introduce a new tailored ontology and integrate it into the **ape!** synthesis process. Secondly, it needs to evaluate the resulting system on various data science use cases. Finally, it needs to compare the results to alternative approaches and discuss the implications of the ontology’s design on the results.

The remainder of this thesis is structured as follows: ?? will briefly overview the core concepts. Next, ?? reviews related work in the field of automated workflow construction and data science ontologies. ?? limits the scope of potential use cases and models the domain ontology. ?? discusses the challenges of using the ontology with **ape!**, presents the transformation of solutions into executable Jupyter notebooks, and introduces an alternative solving backend based on **asp!** (**asp!**). ?? provides an empirical evaluation of the proposed approaches’ effectiveness on the previously defined use cases, compares the performance of the **asp!** backend, and overviews results from experiments with generative **ai!** (**ai!**). Finally, ?? concludes the thesis with a summary of the key findings and an outlook on future research directions.

Chapter 2

Background

2.1 Automated Pipeline Explorer

ape! (**ape!**) [kasalica2022synthesis] is an ontology, synthesis, and constraint-based workflow discovery framework for the automated composition of tool sequences satisfying input-output requirements. It is countering the exploding search complexity common in synthesis tools by limiting its solutions to finite linear workflows.

The semantic knowledge for these is encoded in domain ontologies that contain tool and type taxonomies as well as tool annotations, providing a semantic abstraction layer from the underlying toolsets. Users may then use terms from these to describe their input, expected output, and solution constraints. The latter can be composed with natural language templates, as shown in ?? . **ape!** uses an off-the-shelf SAT solver to find valid workflows and can output executable scripts if the tool annotations contain necessary tool references.

Table 2.1: Selection of APE’s Natural Language Constraint Templates.

Rule ID	Description
ite_m	If we use module \$parameter_1, then use \$parameter_2 subsequently.
itn_m	If we use module \$parameter_1, then do not use \$parameter_2 subsequently.
not_repeat_op	No operation that belongs to the subtree should be repeated over.

It has been evaluated in various domains, such as geovisualization [kasalica2019workflow], bioinformatics [kasalica2021ape], and question answering [kasalica2022synthesis], and even provides a web-based GUI (graphical user interface) allowing users to visually sketch their workflows.

2.2 Data Science

Data Science is an interdisciplinary field combining concepts from statistics, computer science, and its application domain [cao2017data]. It contains concepts such as advanced analytics, **ml!** (**ml!**), data mining, and **nlp!** (**nlp!**) [sarker2021data].

As this field is relatively large, this thesis will focus on a subset defined in ???. Potential workflows in data science may fall into the categories of descriptive analytics, predictive analytics, and prescriptive analytics [cao2017data]. The first uses statistics and visualizations to describe the data, e.g., for reports. The second may use **ml!** to infer patterns and predict future outcomes. The last one uses optimization techniques and targets better decision-making.

While prescriptive analytics is highly domain-specific, descriptive and predictive analytics may be applied to various fields and benefit from automated workflow construction using standard tools. From the different phases of the modeling process [zhang2020data], this thesis will focus on preprocessing, feature engineering, model training, and evaluation. Others, such as data acquisition, labeling, or model deployment, are too application-specific to be included in the workflow synthesis. Hyperparameter optimization and other modeling phases will only be included from a syntactical point of view as their automation is already extensively covered in [he2021automl] approaches.

2.3 Answer Set Programming

asp! (**asp!**) [lifschitz2008answer] is a declarative programming paradigm that first appeared in 1997 with a rich modeling language and is specialized in solving NP-hard, knowledge-intensive search problems using non-monotonic reasoning [Brewka2011]. The paradigm has been used to solve timetabling problems [banbara2013answer], tackle question answering [mitra2016addressing], and even received a controlled natural language (CNL) interface in [guy2017peng]. Its programs often encode tasks following the general pattern of [gebser2016modeling]:

1. Defining the problem domain
2. Generating solution candidates
3. Defining the relevant characteristics of these candidates
4. Testing these properties to remove invalid solutions
5. Optionally optimizing the solutions regarding some defined cost metric
6. Displaying the relevant literals for each solution

These answer set programs may then be combined with the problem instance and passed to a solver to search for valid solutions. The grounder-solver-combination used in this thesis is *clingo* [gebser2008user]. It merges the grounder *gringo*, a tool translating high-level user programs into propositional logic programs, with *clasp*, which finds stable models, so-called answer sets, for these problems. Some features of *gringo*'s extensive input language used in this thesis are facts, rules, integrity constraints, choices, heuristics, and lastly, programs; see ???.

Listing 2.1: *gringo* Input Language Examples.

```
#program check(t).
% Facts
a(0). a.
```

```
% Rules
b(X) :- a(X), not c(X), a.
% Integrity Constraints
:- c(X), a(X+t).
% Choices
{ c(X) : b(X) }.
% Heuristic
#heuristic c(t). [1, true]
```

Choices enable the generation of the search space, which can then be restricted to valid workflows using integrity constraints. Heuristics can control which solutions are found first by the solver, thus directing the search without changing the possible answer sets. The example shows a program called **check**, which is grounded with a constant **t** and directed towards solutions, where **c(t)** is true. The segmentation of programs and partial grounding and solving enables the iterative extension of the search space, increasing the workflow length only if no other valid solution is found at the current limit.

Chapter 3

Related Work

Various ontology-based workflow construction approaches have been implemented in the past. However, most do not target workflows from the general data science domain. **ape!** [kasalica2022synthesis], the framework used in this thesis, is designed for scientific workflows and has been evaluated in multiple fields, including bioinformatics and geology. Some other solutions focus on data mining, a subfield of data science concerned with discovering structures in large datasets [hand2007datamining]. For instance, [OntoDMWorkflowComposition] transforms the users’s input-output requirements into constraints to create a directed, acyclic workflow. Similar to APE, their ontology contains multiple taxonomies: knowledge and algorithms, which correspond to types and tools transitioning between those types.

Ontologies in data science exist. However, they often specialize in specific subfields, like the extensive OntoDM [ontoDM] ontology for data mining, which includes relevant types, algorithms, their components, and tasks, or cover other scientific areas, such as the hundreds of ontologies found on the Ontology Lookup Service [ols4] for bioinformatics. The IBM Data Science Ontology [ibmdatascience] is one of the few general-purpose ontologies for data science. It is based on popular libraries, e.g., Pandas and scikit-learn, and includes concepts and annotations for both Python and R. The content is similar to the one used in this thesis but is not yet in a format fit for the automated construction of executable workflows. Other concepts for standardizing data formats and structures across services, such as the Microsoft Common Data Model [microsoftcdm] or Data Catalog Vocabulary [dcat], lack the detail and context required for workflow construction.

Finally, many auto-ML concepts exist that aim to automate the entire data science process, including data preparation, feature engineering, and modeling [he2021automl, hutter2019automated]. Some assist data scientists, while others target domain experts to develop **ml!** pipelines without technical knowledge. These pipelines, however, are often limited to a single task, the training and tuning of various model architectures, and thus, are not suitable for the general-purpose workflows targeted in this thesis.

Chapter 4

Data Science Domain

Data Science is a large domain with many sub-disciplines [sarker2021data, cao2017data]. Thus, the scope needs to be defined and limited for an effective application and evaluation of APE in this field. Additionally, modeling the ontology requires a specific goal and user definition [noy2001ontology] to select an appropriate tool and type set as well as the applied abstraction level. This chapter defines the selected sub-fields of the data science domain APE is going to be used in, introduces the three use cases for the evaluation, and contains an overview of the ontology, which was created based on these scopes and assumptions.

4.1 Limiting the Scope

In this paper, the data science field is not the target application domain but rather provides a set of methods and tools to transform data and solve problems in other domains. Hence, describing target workflows, which APE will be able to draft, requires defining a subset of data science areas suitable for automated construction and integration into these workflows, as well as the desired user interactions with APE. These areas would contain repetitive and well-defined steps, which could be applied to different problems without requiring structural changes dependent on domain or even problem-specific knowledge. This allows the user of APE to follow the iterative workflow sketching process on the provided semantic abstraction level. Finetuning tool parameters or even implementing structural changes with expert knowledge should still be possible at the end of the construction and exploration process.

4.1.1 Targeted Problem Areas

We select the three areas of **eda!** (**eda!**), simple predictive modeling, and text analysis to demonstrate the capabilities of APE when being applied to data science problems.

1. **eda!** - Exploring given data is usually one of the initial tasks when working with new data [tukey1977exploratory], and thus, does not yet require problem-specific solutions to get most of the desired results. Steps used during data exploration are, e.g., distribution visualizations, missing value checks, or the application of standard transformations [bruce2020practical]. Nevertheless, some tasks, e.g., mapping keys when merging data from different sources,

cleaning data, or deciding on specific ways of handling missing data, might call for knowledge and, thus, tools not contained in the ontology. The extent of necessary changes in the workflow will be evaluated later in this paper.

2. **Predictive Modeling** - Besides producing descriptive statistics derived explicitly from the data, predictive analytics applies knowledge gained from historical data to infer new inputs [cao2017data]. Generic tool sequences may deliver a baseline for later models and could be enhanced with implicit insights found in the given data, problem context, and the user’s domain knowledge. Part of the working hypothesis is that the number of needed adaptations in the synthesis output might diminish the benefits of using APE. Hence, the next stage, prescriptive analytics, which may introduce even less reusability between problems, is not in the scope of this paper.
3. **Text Analysis** - The third area evaluates working on unstructured data with APE. The additional preprocessing steps to extract tabular data may require new complex data or tool structures and, thus, impact the usability of APE with limited technical expertise and reduce the level of detail in the produced workflows influenceable by the user.

Finally, we define the target audience domains for APE in the data science field. The user is expected to either have some statistical knowledge from related fields, such as cognitive sciences, which would allow them to quickly explore workflows and apply the implementation-independent data science practices to a new problem without having to study a new set of APIs, or to have no statistical knowledge at all, but be able to use their specific domain expertise to provide semantical context to the problem data and explain results produced by the synthesized workflow.

4.1.2 User Interactions with APE

The user interactions with APE can be described in three parts: encoding the available inputs, describing the desired workflow with constraints, and inspecting the synthesized workflow. While the second part may be done entirely in the APE system, input encoding and output inspection or adaptation are heavily influenced by the underlying workflow execution layer. For one, the input data is often tabular, and therefore, any use of column-specific attributes, e.g., data type or column key, requires the input schema to be read and encoded prior to running APE. Furthermore, the workflows may produce multiple statistics, visualizations, and tables, all of which need to be presented to the user in a format that allows them to compare and explore the different workflow variants.

To fulfill these requirements, we decided to parse the APE output graph into a Jupyter Notebook, a standard tool in data science, to interactively write code, display results, and document the process. Kernels exist for more than 50 different languages, such as R, Spark, and Python, and can be used to control the execution of individual steps [kluyver2016jupyter]. The latter was chosen for this project as a backend for its popularity and the vast amount of available libraries. However, the ontology and user interactions remain primarily independent of the specific tool set implementations and could be used with another Jupyter kernel. Documentation of the used tools and types in the notebooks could simplify extending code cells and exchanging steps between notebook variations and APE iterations.

With Python as the backend, the tool implementations will mostly be simplified, wrapped versions of methods from existing data manipulation libraries. The Pandas library [mckinney2011pandas] is commonly used to work with any tabular data and can be used with different input formats. Before calling APE, this library is utilized to extract the attributes required to represent the input objects with APE taxonomy terms. In addition to so-called **DataFrames**, which contain tables and their metadata, the library also introduces similar **Series** objects for individual columns and arrays and a collection of commonly used methods to describe and manipulate data. The underlying data in the **DataFrame** and **Series** instances is kept in **ndarrays**, a type from the Numpy library [numpy]. The popularity of the other used libraries, such as Matplotlib [barrett2005matplotlib] and Seaborn [seaborn] for visualizations, scikit-learn [pedregosa2011scikit] for simple modeling, and spaCy [spacy], NLTK [loper2002nltk], and Gensim [vrehuuvrek2011gensim] for natural language processing, will allow for more straightforward adaptations into the APE tool set. Most importantly, the user may use the existing resources associated with these libraries to adapt and finetune the workflows produced by APE.

4.2 Defining Evaluation Use Cases

Three widespread data science problems with openly available datasets were chosen to evaluate APE in the previously defined areas, one for each area. The housing prices dataset [house-prices-Data] evaluates the **eda!** workflow with multiple short tool sequences and many user interactions. Next, the Titanic machine learning problem [titanicData] covers the second area. The simple binary classification task predicting survival evaluates the workflow exploration for feature engineering and model selection. Lastly, to test the limits of applying APE's concept in data science, we chose the IMBD movie review dataset for sentiment analysis [IMBDDData]. Due to its unstructured input and binary classification goal, this problem requires different feature extraction steps for the textual data while keeping the modeling simple. Getting an overview of each use case's tool and parameter requirements enables us to model the data flow and create the ontology.

4.2.1 Housing Prices

eda! is the focus of this evaluation use case. While the other two problems also require understanding the input data, the housing prices dataset has many features with different data types, making it especially interesting for this area. These 81 columns include, e.g., the dependent variable **SalePrice**, the numerical feature **YearBuilt**, and the categorical feature **HouseStyle**. While this regression problem usually requires a model, we will solely focus on the data exploration, which, for this use case, involves tools for data import, descriptive statistics, data cleaning, data transformation, and data visualization. There is no fixed tool order during **eda!** since most of the operations are independent of each other. Nevertheless, for this use case, we decided on the step order shown in ???. Each group contains several activities that will be elaborated on in the next paragraphs. Their order of execution is usually defined in their explanations or displayed in the graph. Optional activities include steps that are typically included in the workflow but technically fall into another area.



Figure 4.1: Housing Prices **eda!** Steps.

Setting Expectations Before the APE run, the user should apply his domain knowledge to formulate a set of hypotheses about the dependent variable and the relations of the independent ones. These may be used to modify the input and remove some of the encoded columns to lower the search complexity in APE. Since the data table is a plain CSV file, the pandas loading function can be used and placed in every workflow as the first step. The following tool sequences largely follow the most voted Kaggle notebook for this area [**housingprices**].

Exploring Dependent and Independent Variables Univariate distributions can be described with descriptive statistics, for instance, value counts for categorical and mean, median, and variance for numerical features. These tools produce a single value for an input column or a series for an entire table. Location and variability should be calculated with skewness and kurtosis or visualized with distribution plots. Similarly crucial for potential models later on is checking multicollinearity, e.g., using correlation matrices. Multivariate relations will be explored using tools like pivot tables, crosstabs, and relationship graphs. Correlation matrices, pivot tables, and crosstabs will change the column or row index and require additional type taxonomy terms to reflect this and keep tool compatibility. The graphs could be customized in size, style, color, etc., requiring specific functions or interchangeable keyword arguments. At this point in the APE workflow, the user may try to explain the produced results in the notebooks using their domain knowledge and adjust the hypotheses accordingly.

Handling Missing Data Part of cleaning the given data is deciding how to handle missing data. An important factor is the pattern of the missing values. Determining whether the process is random or biased requires the user to inspect descriptive statistics, filtered table views, and possibly visualized indicator masks. As a result, additional aspects must be reflected in the ontology: Filtering should provide a temporary view of the table, and masks must be the same size as tables to be processed together in operations. Other factors include the number or percentage of missing values in each column, the correlation of these features to the dependent variable, and whether a highly correlated feature with fewer missing values can replace it. If the user decides to drop entries or features, the table index will change, and the resulting memory state in APE needs to encode its incompatibility with the previous version. Alternatively, in some cases, instead of removing table parts, the data can be imputed with simple scalar values, such as mean, median, and mode, or with a model that uses the other feature values to estimate the missing ones. The latter will use the scikit-learn libraries transformers, which have similar technical requirements as the models in the subsequent use case.

Exploring Outliers Outliers are another part of cleaning data, and in this use case, they will be handled by inspecting the variable distribution, potentially normalizing it, e.g., centering or scaling, and then deciding whether to remove the entries or not. The process is done for dependent variables before applying the gained knowledge while handling the independent variables. This may be iterated until no outliers are removed. Tools could require threshold values and potential default values for dropping entries, which would also manipulate the table row index. The last areas of data cleaning in this use case that involve changes in the table

metadata are handling duplicate values and casting data types. While the first will remove items from the row index, the second will change the table schema.

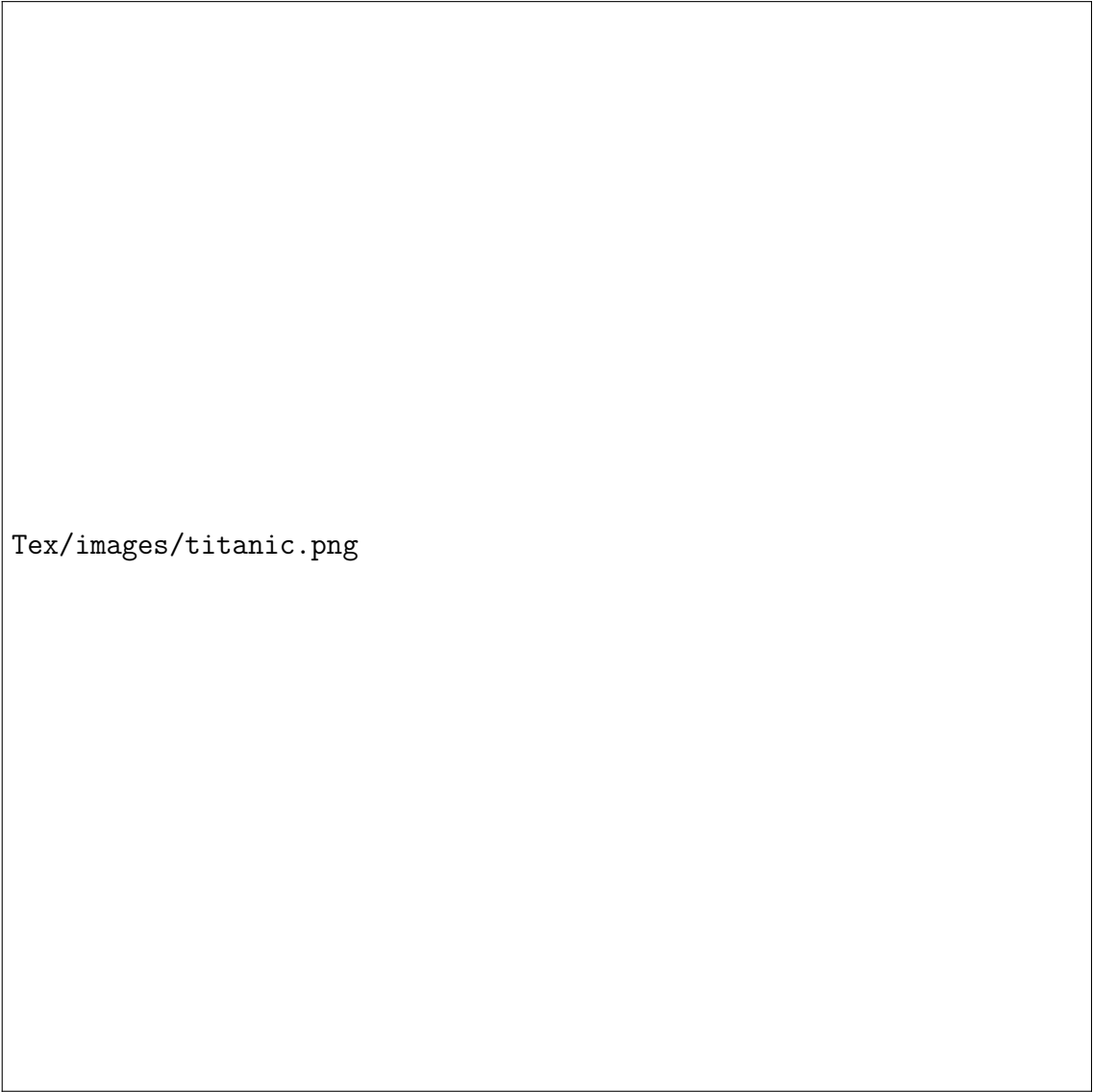
Test Statistical Assumptions Finally, at the end of data exploration, the user might want to test the basic statistical assumptions before training and using a model. We will look at normality, homoscedasticity, linearity, and the absence of correlated errors. The normality of errors in the dependent and independent variables may not necessarily be an issue in large enough data samples. Distribution plots and Box-Cox transformation tests can be used to inspect these before applying appropriate functions if required, such as logarithm, inverse, etc. A new feature might be required to indicate zeros for log transformations. Homoscedasticity, the homogeneity of variance, and linearity can be checked by plotting the features in question, e.g., with scatter plots, and corrected with the previously mentioned transformations and resampling, which would also modify the table index.

4.2.2 Titanic

The Titanic dataset's task is a binary classification problem in which the target variable indicates whether a passenger survived the infamous accident. The inference can be made based on the 11 other columns in the table, such as **Pclass**, **Sex**, **Age**, or **Fare**. While **eda!** is usually part of this modeling use case, the evaluation will focus on preprocessing, feature engineering, model selection, training, and evaluation. Optional steps, which could have technical implications regarding the ontology, are hyperparameter optimization and stacking or ensembling, see ??.

Similarly to the previous use case, the user is expected to have a set of hypotheses about the relationships between the various columns. However, due to the different focus, this time, decisions regarding cleaning data are already made before the first APE iteration. We will again try to recreate the data flow from one of the most-voted Kaggle notebooks [**titanic**]. The **Ticket**, **Cabin**, and **PassengerID** features will be dropped due to high duplicate or missing value percentages or low correlation to the target variable. Moreover, the **Name** column will be used to evaluate string transformations by applying regular expressions to extract the title. Newly created or transformed features will include a total count of family members on board, a single traveler indicator, and an ordinal age feature. Since the number of columns is relatively low compared to the housing price use case, dropping features before calling APE is less about lowering the complexity for the solver and more about simplifying interpreting the APE outputs.

Feature Engineering Creating new columns will modify the table index and could introduce information that is only available at APE or even Python runtime. Further difficulties during feature engineering may occur while encoding categorical or ordinal features. Namely, replacing nominal with numerical values requires extensive domain knowledge and **eda!** results, while one-hot encoding them introduces many new unnamed columns. Correspondingly, binning categorical or numerical into new categorical or ordinal features involves a lot of user interactions. It is expected for APE to draft the tool sequence and for the user to test and evaluate parameter combinations.



Tex/images/titanic.png

Figure 4.2: Titanic Predictive Modeling Steps.

Model Selection and Training The modeling tools will primarily use the scikit-learn classes and functions. Although the titanic problem only requires a classification model, the ontology should also include structures for other types, such as regression or clustering. Model selection relies either on the user’s statistical knowledge or on comparing model evaluations against each other and the baseline model, for which we will use the library’s random estimators. Relevant to this problem are supervised learning estimators such as random forests, k-nearest neighbors, or support vector machines, all of which have different parameter sets. As a result, the ontology will model the shared interface only and leave the remaining parameters up to the user. To correctly estimate the risk of models, the dataset needs to be split into training and test or evaluation sets. The associated tool will split the input table into four objects with different indices: feature table and target **Series** for training and test sets. If **eda!** were part of the use case, splitting would occur before the first exploration step to avoid contaminating the modeling process.

Evaluation Evaluating the model is done by using a set of appropriate metrics. Since the used models are classifiers, we will use scores such as balanced accuracy, weighted f1, precision, recall, or AUC-ROC. These can be used to estimate the risk of the result of one or multiple train-test splits, e.g., by using cross-validation to achieve a less positive bias. Similar to hyperparameter tuning with grid or random search and ensembling, this requires model and parameter inputs of indefinite length. Grid search would require a set of user-selected parameters to tune and values for each parameter. Correspondingly, ensembling could take an arbitrary amount of models as input to combine them into a new one with voting or stacking. The possibilities of arbitrary user input will be discussed in more detail in the following section and the next chapter. The user may iterate the evaluation process for different models and feature sets by iterating the APE process and adapting the constraints defining the model or by fixing a tool sequence but describing the model with nonterminal taxonomy terms, hence allowing APE to generate the desired workflow variants in one call. The final step inserted by APE could be an inference tool allowing the user to predict new labels with the just-trained model.

4.2.3 IMBD Movie Reviews

The last dataset used to evaluate APE’s capabilities in the data science field is the IMBD movie review dataset for sentiment analysis [**imbd**, **basicnlp**]. It is a binary classification problem with labels indicating positive or negative reviews and just one feature: the review text. As previously mentioned, the unstructured nature of natural language texts requires different preprocessing tools, and referencing these and their results requires, in turn, different taxonomy terms. Data import and predictive modeling steps will not differ from the first two use cases. Furthermore, while **eda!** will require different tools for natural language texts to, e.g., identify word and sequence lengths, character sets, or unusual vocabulary, user interactions with APE would be nearly identical because of similar tool signatures. Hence, this part of the evaluation will focus on the text feature extraction tools and the user input as shown in ??.

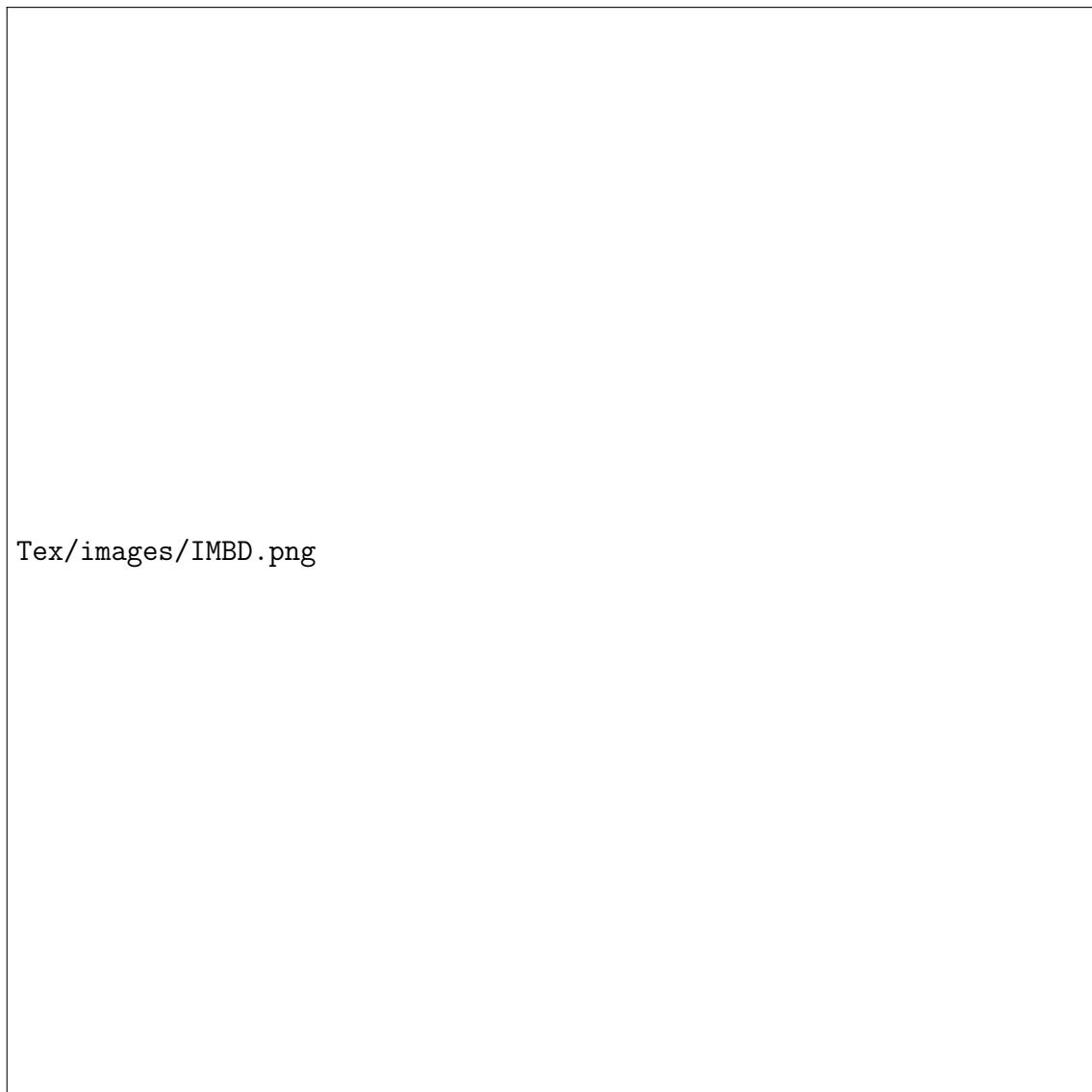


Figure 4.3: IMBD Movie Reviews Text Analysis Steps.

Text Preprocessing For this purpose, we will use both simple and semantic-aware string processing tools. HTML tags and other formatting-related markup elements can be removed with the BeautifulSoup library. Unwanted characters or patterns, such as URLs, will be replaced by matching with regular expressions. Finally, we could use dictionaries to replace domain-specific abbreviations. Semantic-aware tools allow us to transform words and sequences based on language-specific attributes. For instance, the spaCy English language models will remove stop words - common words with low semantic importance - and lemmatize the remaining ones, which reduces each word to its base form. Alternatively, the stemmers from the NLTK library could be used to shorten the tokens. Another popular library to transform words is Textblob, which will be used for spelling correction.

Embedding Subsequently, the preprocessed reviews must be embedded into numerical vectors before models can be trained. The text feature extractors from the scikit-learn library can create embeddings based on word counts and the given vocabulary, a so-called bag of words. A more complex embedding can be applied with the Word2Vec model from the gensim library, which places each word in a high-dimensional vector space learned from the training data text corpus. Both methods will create large matrices as outputs sharing the same row index as the input table but with a column count dependent on the review texts' content. The latter is unavailable at the APE runtime and should thus not be regarded while modeling the ontology. However, the extent to which the user can still interact with these produced types without requiring substantial changes in APE compared to the other two use cases will be a prominent topic during the evaluation.

4.3 Modeling the Ontology

By defining the problem, identifying use cases, and extracting requirements, we implicitly answered vital questions whose answers will steer the ontology modeling process [noy2001ontology]. Namely, we specified the purpose to be a semi-automated search, where the user would iterate the synthesis process and adjust the configuration and constraints. Furthermore, we expect the resulting workflow to be executable but not necessarily optimal. Additional crucial aspects discussed in this section are the level of detail in the taxonomies and the differentiation of tool inputs vs. parameters, which we have already touched upon during the use case introductions. Heuristics are currently not natively supported by APE and, therefore, not mentioned here. However, we will introduce them into the ontology when changing the backend of APE in ???. Finally, the collaboration and social contract aspects of this new domain ontology are not covered here as we mostly inherit the advantages of the APE ontology format. The only difference is that the taxonomy structures are primarily adopted from the used tool libraries' class hierarchies, which may simplify working with the ontology for those already familiar with these standard data science tools. The following subsections will provide an overview of the tool taxonomy, the type taxonomy with its dimensions, and their interactions in the tool annotations.

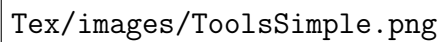
The image area is mostly blank, with the text 'Tex/images/ToolsSimple.png' located in the lower-left corner. This likely represents a missing or placeholder image for the figure.

Figure 4.4: Selection of Tools from a Simplified **eda!** and Feature Engineering Sub-Taxonomy.

4.3.1 Tools Taxonomy

The tools needed to synthesize workflows described in the three use cases can be separated into the six overlapping sub-taxonomies **EDA**, **Feature Engineering**, **Plotting**, **Modeling**, **Encoding**, **Embedding**, and **Utility**. In contrast to the type sub-taxonomies, the tool hierarchies are primarily based on their semantic effect instead of their associated libraries, especially since many of the used libraries share functionalities. For instance, the scikit-learn library provides a tool to plot their decision tree objects as a graph. While most of the 113 leaves in the tool taxonomy are implemented as wrappers for common library function call sequences, some are required to ensure compatibility with APE, namely, those in the Utility sub-tree.

The first sub-taxonomy for **eda!** and simple feature engineering primarily contains methods from the Pandas library. A representative selection of tools can be seen in ???. The implementations of this sub-taxonomy allow for table and column description, handling missing values, string manipulation, pattern extraction, ta-

ble reshaping, and column transformation. Even though the transformer objects are part of the scikit-learn library and share similarities in their interface with the models, they are categorized into this sub-taxonomy as these classes are intended to modify tabular data. The figure shows how, in addition to being organized by their purpose, e.g., table summary statistics or string operations, the tools are also agglomerated by their operation type, such as whether they manipulate an entire table or just one column, whether the row index is getting modified, and if the operation is in-place. Consequently, each tool may be a child node of multiple parent nodes, all of which can be used by the user to describe the desired tool. For instance, a tool state node constrained by `TypeConversion` and `MissingData` could be instantiated by the leaves `isna` or `notna` in APE.

Next, the Plotting sub-taxonomy comprises tools for data visualization functions from the Seaborn, scikit-learn, and Wordcloud libraries. They cover tasks like graphing variable distributions, relationships, and trends, as well as more complex graphs, like tree feature importance plots and word clouds. Additional boilerplate function calls were added to the implementations to enable figure customization, such as setting graph size or rotating labels. These aspects can be controlled with optional keyword arguments through different tool modes in APE or by the user while fine-tuning the workflow. Together with the previous sub-taxonomy, these tools fulfill the majority of the housing price use case requirements.

While the scikit-learn library is also used for data transformation, its primary purpose in this ontology is the enablement of classification workflows as outlined by the Titanic problem. Indeed, all model classes stem from scikit-learn, as do the helper functions for the other workflow steps: model selection and evaluation. However, due to the SAT problem encoding in APE, accessing features and target labels separately, as required by these models, proves difficult. This issue of handling tables, a dominant type in data science, is one of the main challenges in this paper and will be discussed in detail in the following sections and chapters. Here, utility functions are added to control the flow of tabular data in APE in the training and prediction steps.

Similar issues attributed to the tabular structure occur when encoding nominal or ordinal features. Both steps are essential since the used model classes require all inputs to be numerical and are implemented by extended Pandas functions. Furthermore, the text analysis use case introduces the need for text embeddings, vectorized representations of unstructured data that models can handle. Like the encoding tools, vectorization will modify the table column index, and the resulting shape is unknown at APE runtime. Moreover, in most cases, the embedding table has such a high column count that dimensionality-reduction tools are deployed to lower the complexity before further analysis. They are another set of tools that will modify the table index and are implemented by scikit-learn transformers in this ontology.

4.3.2 Type Taxonomy

To properly represent the different data types in our use cases, we utilize APE's capability to handle multiple dimensions in the type taxonomy. In addition to the primary dimension `DataClass`, the smaller sub-taxonomies `DataState`, `Statistical Relevance`, and `DataSetIndex` describe each type state in the workflow synthe-

sis. The advantages and disadvantages of adding these optional dimensions will be analyzed in the evaluation ??.

Primary Dimension - Data Class

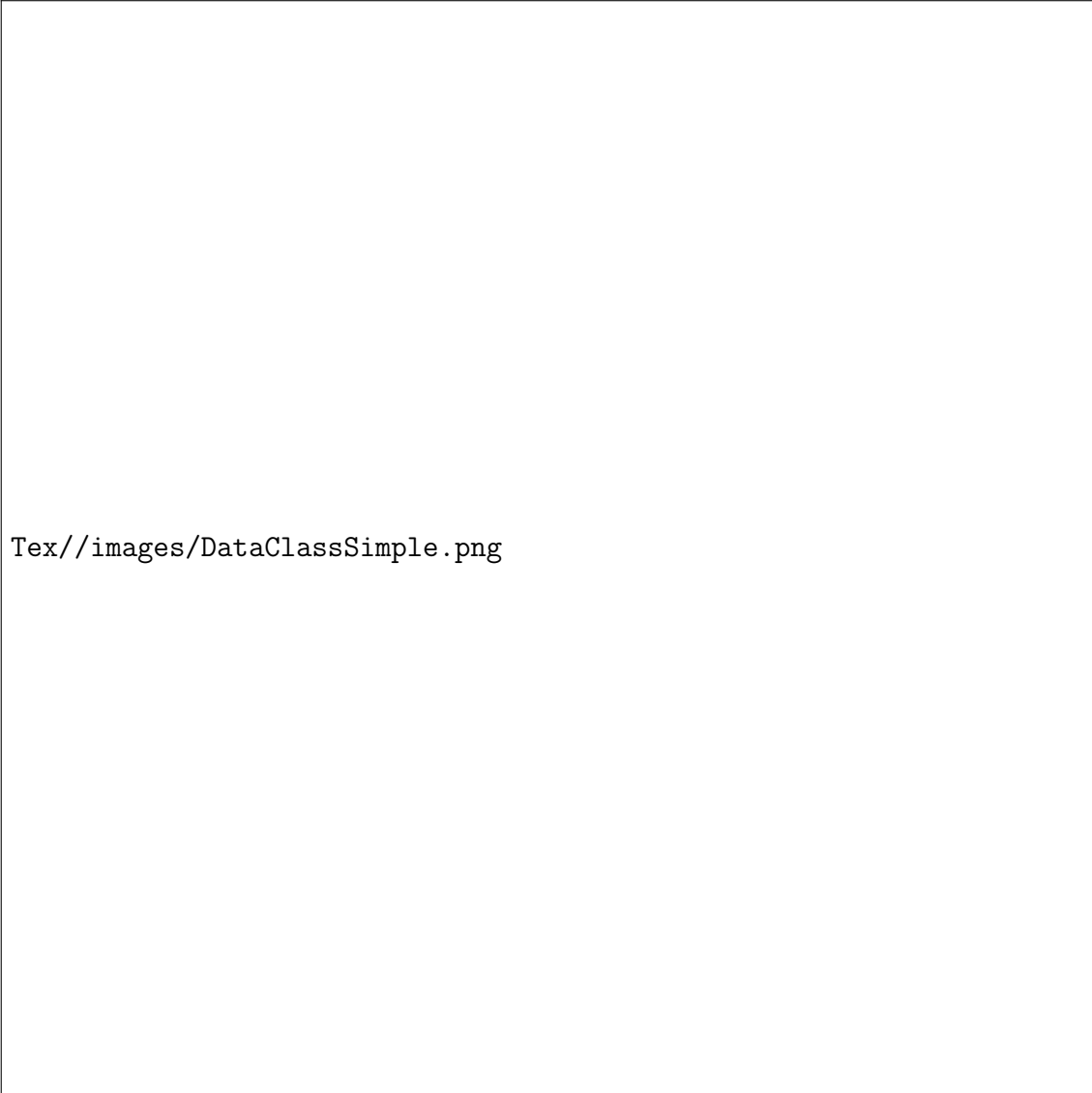


Figure 4.5: Selection of Types from the **DataClass** Dimension.

As visualized in ??, the types in the primary dimension are categorized into six disjunct sub-taxonomies: **BasicObject**, **PandasObject**, **MatplotlibObject**, **NumpyObject**, **SklearnObject**, and **TextEmbeddingTransformers**. While the first five hold types from their respective libraries, the last sub-taxonomy consists of various text-processing transformers. Here, multiple inheritances were only required for the scikit-learn objects, which are grouped by model and target label type, e.g., **DecisionTreeClassifier** being both **SL** - supervised - and **Tree** - a tree model.

Basic types can either be outputs of descriptive statistic functions or user-defined custom inputs passed in the configuration or constraint file. While their value does not influence the workflow synthesis process, they are later required for notebook construction. For this reason, we are using APE's implicit dimension **APElabel**,

which is created at runtime from all the `APElabel` values in the input type nodes. For instance, if an input of type `Int` had the label `42`, that value may get passed along to a tool accepting a parameter of type `Int`. The limits of this method will be discussed in the following subsection and chapter.

Additional potential problem sources are the complex types, especially the tables implemented here as Pandas `DataFrames`. These tables consist of multiple columns, all of which may hold different basic types. Some tools require the entire input table to have a specific type. In contrast, others produce tables with a particular format or type, such as correlation matrices, which always contain numbers, or classification reports, which have a fixed column index. In response to these inherent requirements, we use detailed `DataFrame` object types and add two more `PandasObject` subtypes: `Column` and `Series`. To address columns individually, each column in a table is encoded into a type node as follows: A table with a column containing strings would be encoded into the detailed table type node and a `StringColumn` node. If a column is separated from a table and write operations do not modify the source table anymore, its `DataClass` type changes to `Series`, e.g., when the input table is split into a feature `DataFrame` and a target label `Series`. At the Python layer, values in these Pandas objects are stored in Numpy arrays. However, to avoid the decrease in technical abstraction, the only Numpy type used in the three use cases is `EmbeddingMatrix`, which is required for text feature extraction.

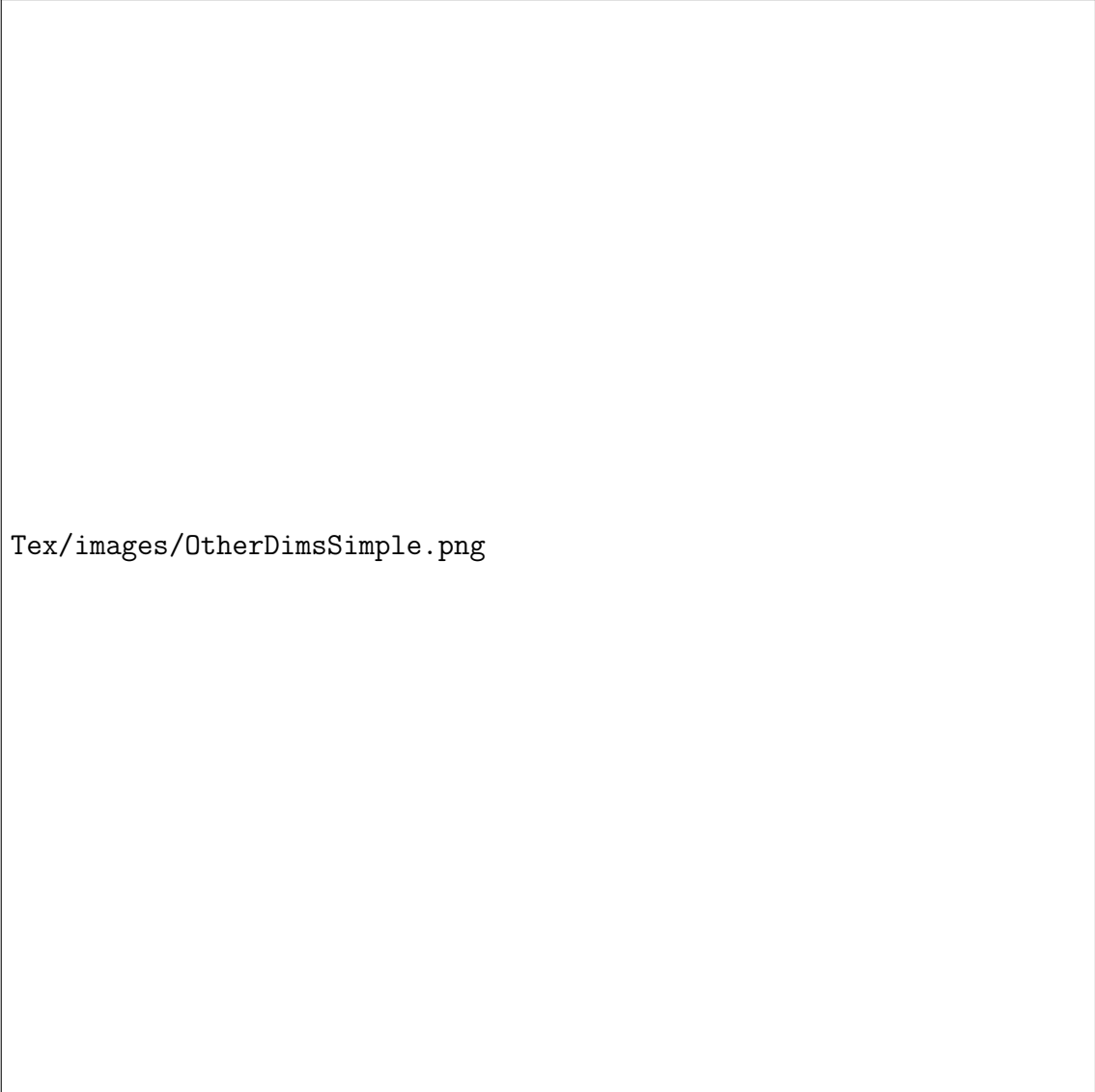
The visualizations produced by the previously mentioned libraries are stored in Matplotlib objects, namely `Figures` and `Axes`. Each `Axes` object can hold a plot with its metadata and can be used to set, e.g., label orientation or scale. `Figures`, in turn, contain one or more `Axes` objects and control aspects such as layout and overall size. As a tuple, they are the standard output signature for most of this ontology's plotting tools.

Finally, the `SklearnObject` and `TextEmbeddingTransformers` sub-taxonomies include complex feature engineering and modeling classes. Namely, instances from these collections differ in their requirement to be trained or fitted before any data transformation or prediction is possible; see ???. The data class `Sklearn Object` comprises dummy estimators and various classification, regression, and clustering models. Including different architectures and baseline models enables assessing the exploration potential of APE in a modeling workflow. On the other hand, `TextEmbeddingTransformers` has only three concrete transformers due to the text analysis use case focusing on the feasibility of interactions with the produced output structures instead of workflow exploration or exploitation.

Secondary Type Dimensions

Each secondary dimension adds information already available in the input data or inferable by APE during the synthesis to increase the probability that the resulting workflow is syntactically and semantically correct. However, applying this level of detail is optional, and the user may always utilize default values, such as the root type `DataState` for any data state, including none, or `NoState` for explicitly none, as seen in ???.

Data State This dimension is required for scikit-learn objects. Namely, it acts as a flag differentiating fitted and non-fitted models and transformers. Since some



Tex/images/OtherDimsSimple.png

Figure 4.6: Secondary Dimensions with fallback types.

tools are only available for one of these states, adding this information avoids non-executable scikit-learn function calls. Furthermore, the data state dimension allows users to wield more fine-grained control over Pandas and Matplotlib objects. If feature engineering is part of the workflow, one requirement may be to normalize numbers and clean textual data, and **Figures** may only be accepted as output if they are filled with a plot. Type nodes of other data classes have no dedicated data state values and, thus, use the **NoState** type.

Statistical Relevance The next dimension is optional for syntactical correctness. However, adding its types to the ontology provides more detail to the tool annotations and, as a result, reduces the number of required user constraints to reach the desired workflow. Statistical relevance primarily applies to Pandas objects and describes their role in a modeling workflow. Moreover, these types can also be applied to objects from the scikit-learn library to specify their purpose. For instance, a transformer or estimator may have been fitted to impute or predict dependent or independent variables. Other data classes do not have specific types in this dimension

and share the `NoState` type.

Dataset Index Finally, the dataset index dimension assists in making sure that steps using tools accepting two or more tables share the same table index and, hence, are executable. In a modeling workflow, a feature `DataFrame` may be transformed and cleaned. However, if some index modifying operation, such as dropping duplicates, was applied, the resulting `DataFrame` is no longer compatible with the original target label `Series`, and the notebook, while syntactically correct, would raise an error at runtime. The available index types are split into original indices, derived directly from the input data or during the modeling workflow, and temporary indices, created during data exploration for a specific purpose, such as a correlation matrix. In addition to being available to Pandas objects, these types can be applied to Matplotlib and scikit-learn objects to document their data source. The `NoState` type is assigned to objects of the remaining data classes.

4.3.3 Tool Annotations

Modeling the ontology is an iterative process, and each time a new set of tools or types is added, the tool annotations, which combine these taxonomies, are potentially subject to modification. These derived transition rule changes efficiently reveal conceptual flaws in the ontology, as we will show in ???. The tool signatures closely resemble those of standard data science libraries, with the table access concept as the central assumption and starting point for tool modeling. Since the Pandas library has an integer position and label-based row and column indexing system[mckinney2011pandas], any tool in the ontology that accesses a specific part of a `DataFrame` also requires the index parameter. However, creating entries for each row or integer position in a dataset is impractical. For this reason, APE-synthesized table access relies on column labels. For instance, this simplified signature for a scatterplot (`DataFrame`, `NumberColumn`, `NumberColumn`) -> (`Figure`, `Axes`) illustrates how APE represents the parameters for the source table and the x and y value columns.

Currently, most leaves in the tool taxonomy are overloaded in their implementation. These differ in their optional parameters, such as graph coloring or styling, but also regarding parameter types, as exemplified by the scatterplot function. In addition to two columns for x and y values, it accepts column label parameters for marker hue and style, as follows:

```
(DataFrame,NumberColumn,NumberColumn,Column) -> (Figure,Axes)
(DataFrame,NumberColumn,NumberColumn,Column,Column) -> (Figure,Axes)
```

APE unifies these rules during the workflow search, and each term is assigned a terminal type according to the taxonomies. Consequently, the type states corresponding to the first three parameters in the scatterplot signature could be unified with their terminal `DataClass` sub-types, such as

```
(MixedDataFrame, IntColumn, FloatColumn)
```

However, the SAT solver still requires explicit state transitions despite APE allowing implicit type and tool states in its input files. As a result, any implied relation between a tool's input and output parameters must be made explicit. If, e.g., a tool

changes the `DataState` of a `SklearnObject`, a non-terminal `DataClass` type, from `NonFitted` to `Fitted` the implied unifications should have the following form:

```
((X, NonFitted), ...) → ((X, Fitted), ...)
  where X is terminal sub-type of SklearnObject
```

A preprocessing step is introduced to circumvent this lack of generic type variables in APE tool signatures, transforming these implied input-output transitions into explicit terminal rules. Notably, for this ontology, the instantiation step following generation and constraint patterns, not unlike a declarative solver, produces approximately 1000 tool modes from around 100 tools.

4.4 Discussion

This chapter defined the targeted problem areas and users for this ontology. Next, three use cases were selected to evaluate **eda!**, predictive modeling, and text analysis. Based on the derived requirements, we modeled the ontology consisting of tool and type taxonomies and tool annotations. However, the obtained result is a mere starting point representing everyday data science tools and types. It is meant to be extended for additional use cases and adapted to fit technical requirements.

At the current stage, the ontologies functionality is still theoretical when combined with APE. Potential challenges include the complexity of the created SAT problems and the usefulness of the various APE outputs. The latter is because many overloaded tools could lead to all found workflow variants being structurally identical with slight parameter changes. Furthermore, the scalability of this ontology is questionable: On one hand, the taxonomies roughly imitate the class hierarchy of each used library and should be navigable with data science knowledge. On the other hand, the applied technical abstractions do not follow an existing common standard. Thus, it might be challenging for various parties to maintain these novel taxonomies with their multiple inheritances.

The following two chapters will introduce two approaches to using these ontologies. They will address existing and new challenges arising from each course and apply the required modifications to the ontology. However, other aspects, such as maintainability, will not be evaluated and are subject to future work.

Chapter 5

Native APE in Data Science

APE has been used to synthesize and explore scientific workflows in numerous use cases, such as geovisualization and question answering [kasalica2022synthesis]. However, applying the framework and its concepts to the field of data science presents a novel set of challenges, some of which were already briefly mentioned while modeling the ontology in ???. This chapter delves into the details of extending the APE workflow and adapting the data science ontology to achieve the set target of executable Jupyter notebooks.

First, we will examine the difficulties of combining the domain and tabular-data-specific requirements with APE’s synthesis concept to construct workflows with lower-level tools that will run in a general-purpose programming framework. These include the higher-than-usual dimension and input count, the direct management of data references by the user, and the data flow handling by tools in APE’s encoding.

Next, we will revisit the data science ontology, focusing on the modifications for a final version compatible with native APE. The section will give a short rundown of the new tools and slimmed-down type taxonomy while offering insight into the rationale behind these changes.

Finally, we will discuss the implementation details of how the APE workflow was adapted to fit the data science domain better. Here, we introduce a new input data preparation tool, the APE output parser, and the library of wrapped Python tools that facilitate the automated creation of executable Jupyter notebooks.

5.1 Challenges

Combining APE with the data science ontology and use cases defined in ??? with no adaptations leads to numerous challenges regarding SAT search, workflow executability, and user interactions. The reason for this lies in the SAT encoding of the workflow search, which, as will be shown, is unsuitable for working with tabular data in some aspects. This section will overview the most prominent issues and discuss potential workarounds where possible.

5.1.1 Workflow Encoding

To gain a basic understanding of the ontology-based synthesis process, we will examine the workflow formula of the SAT encoding in ??:

$$\begin{aligned} \llbracket W \rrbracket_n := & \bigwedge_{i=1}^n \left(\bigvee_{op \in L^\circ} op(m_i) \right) \\ & \bigwedge_{i=0}^n \bigwedge_{j=0}^{k-1} \left(\bigvee_{ty \in L^t} ty(in_i^j) \right) \\ & \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{l-1} \left(\bigvee_{ty \in L^t} ty(out_i^j) \right) \\ & \bigwedge_{i=0}^{n-1} \bigwedge_{j=0}^{k-1} \left(\epsilon(in_i^j) \vee \bigvee_{p=0}^{i-1} \bigvee_{q=0}^{l-1} Bind(in_i^j, out_p^q) \right) \end{aligned} \quad (5.1)$$

APE encodes a workflow $\llbracket W \rrbracket_n$ as a sequence of n steps, each having a set of inputs and outputs. Within this encoding:

- Each step i uses a tool op from the ontology. An auxiliary encoding enforces the tool and type taxonomy properties, such that using a non-leaf node implicates the usage of one of its child nodes and vice-versa.
- Input and output parameters are so-called type states ty , defined by their type dimensions. APE uses the same number of parameters with every tool and adds empty type states ϵ for tools with fewer.
- Any non-empty input type state can be bound to an output type state of a previous tool, ensuring the dependencies in the linear sequence. To add external IO, workflow inputs and outputs are added as the 0th tool's outputs and the $n + 1$ th tool's inputs.

5.1.2 Input Labels

Since type states derive their values from the dimensions defined in the ontology, a tool parameter can only adopt values already defined in the type taxonomy. Hence, it is impossible to use constraints to introduce arbitrary inputs to the workflow, such as new column labels or numerical graph sizes. As a result, selected values must be added to both the taxonomy and workflow inputs before the user can utilize them in constraints. This can be done without changing the ontology by using `APE_label`, an implicit type dimension created by APE during runtime with an empty default label. Any values in the workflow input listed under that dimension, as illustrated in ??, are appended to the sub-taxonomy and can then be referenced during the synthesis process and the workflow execution.

Listing 5.1: Example of APE IO Configuration.

```
"inputs": [
  {
    "DataClass": ["StrColumn"],
```

```

    "APE_label": ["Name"]
  },
  {
    "DataClass": ["Int"],
    "APE_label": ["42"]
  }
],
"outputs": [
  {
    "DataClass": ["Figure"]
  }
]

```

However, because APE adds these labels during its runtime, none appear in any tool transition rule. Consequently, no tool produces labeled outputs, and type states with an explicit `APE_label` value only exist as workflow inputs. Modifying the tool annotations to account for this new dimension with potentially as many values as inputs would increase the search complexity by a drastic amount, as shown later in this section.

5.1.3 APE Data Flow

Moreover, the *bind* predicate may assign any already generated outputs to a tool input, even if they have already been used earlier. Instead of message passing, this memory bus architecture leaves inputs available for reuse without needing to pass them on explicitly. While this may simplify the tool annotations, it also leads to various problems in the data flow when combined with the loss of `APE_label` values after any operation. Three notable examples of affected transformations are:

- **nonfit** \rightarrow **fit**: Training a model in an APE-generated workflow results in two models, even though the unfit model does not exist anymore. In the executable program, both references would point to the trained model. Making the tool annotation in-place by removing the output would also remove any metadata the fitting operation might add, namely, which data type it is predicting or what data it was trained on. That information is available at APE runtime and may be crucial in creating a valid tool sequence but could not be passed on.
- **str** \rightarrow **int**: Type casting a column should modify the `DataClass` and keep all other dimensions intact. However, a transition rule doing that would also remove the column label and introduce a copy of the input column. Even with the latter being a potentially intended effect, having no label on the copies would make them indistinguishable after two type casts with the same output type. By contrast, an in-place transition rule with no outputs could lead to cast results with incorrect type dimension values. Hence, tools would have to accept wrong input types, e.g., numeric models accepting string columns that could have been cast already, which would lead to more tool modes and, thus, higher search complexity.
- **(DataFrame, Column)** \rightarrow **(DataFrame, Series)**: Modeling might require removing the dependent variable from the table and, later on, splitting the

data into train and test sets. Neither operation can be implemented to operate in-place since that would lead to so-called data leakage and the potential invalidation of any conclusions. However, the label loss of the non-in-place functions prevents the best practice of starting the workflow by splitting the data to counter data leakage.

A lack of labels would lead to the non-addressability of any processed inputs and, as a result, drastically change the typical way of working tabular data to explicitly chaining tools with constraints. For this reason, the native APE version of the tool ontology will proceed with in-place operations. Therefore, the evaluation will primarily focus on two aspects: the effects of missing type changes on the synthesized workflows and, more importantly, the impact of missing outputs on the generation of tool sequence dependencies.

5.1.4 Search Complexity

Finally, we will analyze the search complexity with respect to the number of inputs, workflow length, and number of dimensions in our type taxonomy. A set of n unique tool usage constraints forces APE to generate at least n steps. We choose 20 as the upper bound with an unrealistically high timeout of 10000 s to ensure an out-of-memory error resulting from the search complexity. The housing prices data set is chosen for its high number of inputs. Sixteen experiments are run: 1, 2, 3, and 4 dimensions¹ with approximately 100%, 50%, 25%, and 12.5% of the available columns.

The graph ?? shows the seemingly linear relationship between the number of generated clauses and the workflow length. Interestingly, all experiments run out of heap space except for the configurations with three dimensions and ten inputs, and all runs with four dimensions, which timed out first.² From this, we can infer that the memory usage of APE and its solver scale with more than just the number of clauses. Another observation is the significantly increasing complexity with rising input or dimension counts. The first relation is at least linear, while the second seems exponential, which is troubling since two dimensions already seem to limit the workflow length to eight, and using all four of them seems to cause a timeout or out-of-memory error after just one step.

Looking back at ?? assists in explaining the linear and exponential relationships. The top-level conjunctions scale linearly with the workflow length and the fixed tool parameter counts. Similarly, the inner disjunctions on unary predicates scale linearly with the number of possible types. Only the disjunction on the binary bind predicate would exceed linear growth. This fact is, however, made insignificant for the relatively low workflow length by the much larger sizes of the tool and type taxonomies shown in ?. The graph also indicates their exponential relationship

¹The APE_label dimension is implicitly created on every run and, hence, is not accounted for in this experiment.

²The experiments were run on an Apple M1 Max with 8GB of assigned Java heap space. The timeouts on the runs with ten inputs occurred after roughly three and nine hours. The others did not finish setup for step one within more than double the given time limit and had to be interrupted externally. Run times longer than the stated 10000s might have been possible due to potentially missing timeout checks in the solver itself.

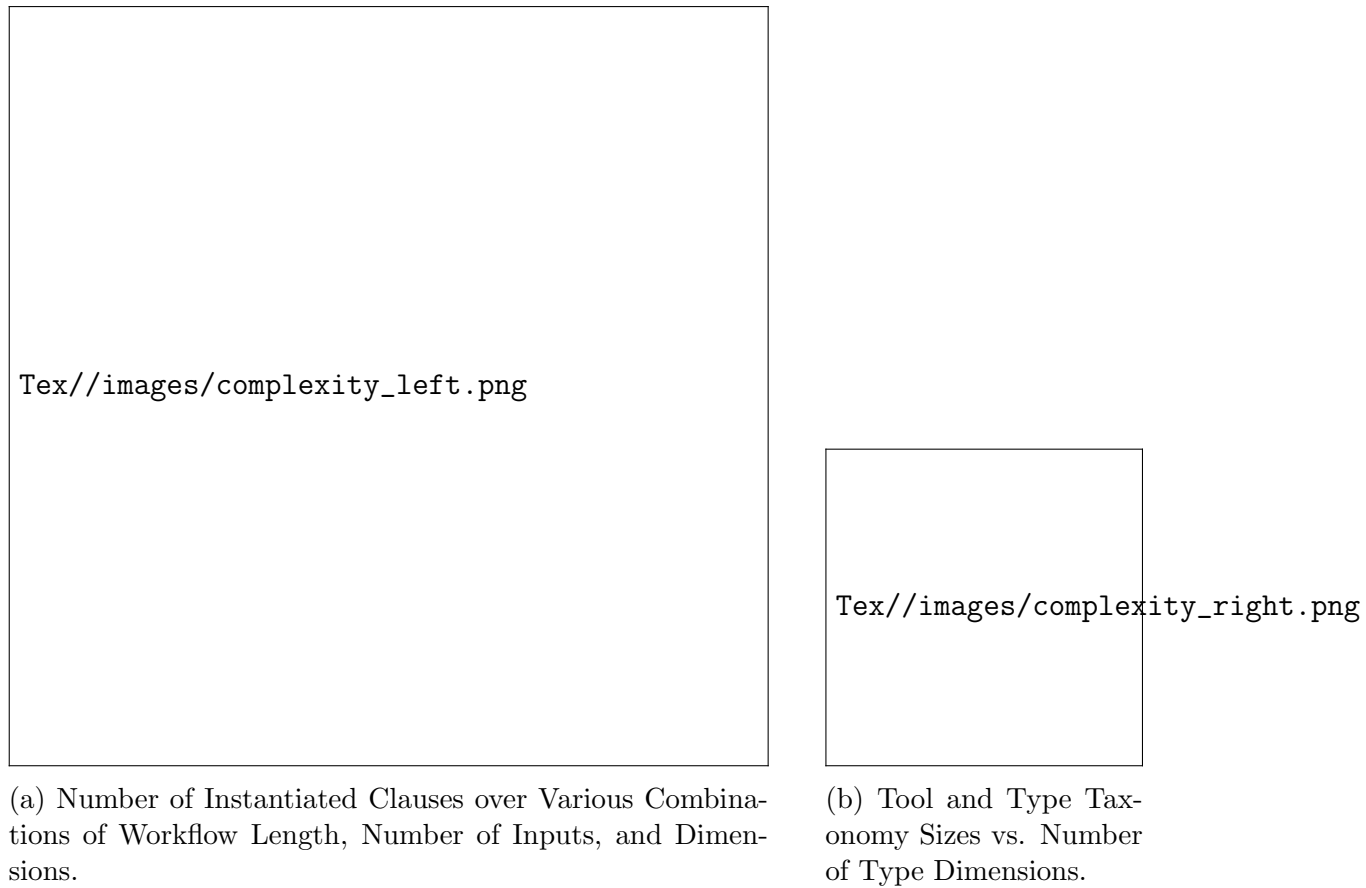


Figure 5.1: Measurements from Search Complexity Experiments.

with the number of dimensions³, which carries over into the number of generated clauses.

5.2 Ontology Adaptation

Addressing the in-place tool and complexity challenges requires the modification of the data science ontology. This subsection examines the changes in both the tool and the type taxonomy.

5.2.1 Tool Taxonomy

First, we will overview the new modes added to the toolset, reflecting the in-place data flow and reducing the overall number of tools required for modeling workflows. Notable affected tool sub-taxonomies include:

- **Plotting:** These functions are kept as in-place tools since no labels need to be passed through. Usually, the inputs contain a table and various column selectors to produce a new plot instance. However, there could be problems

³Adding one dimension potentially adds multiple new subtypes to every existing type, thus increasing both the number of types and explicit transition rules exponentially.

when further customizing a figure, e.g., setting the figure size or label orientation. Some plotting data types must also be included in the tool inputs in these cases.

- **Estimators:** The fitting and prediction tools also continue to operate in-place. The state change from non-fit to fit is essential for implicit tool ordering. And even though model outputs could benefit from labels and data types, changing the tool to operate in-place would overwrite the original data.
- **Type casting:** Commonly, type casting would be done in-place, but the workflow constructed by APE would not be able to reflect the change in the type state. Hence, a non-in-place version is chosen that will lose the label but assign the correct type.
- **Data Splitting:** Both the non-in-place and the in-place versions are kept. However, neither can pass on a label to the new series split from the input table. The same happens when train-test-splitting: The non-in-place tool produces two new tables and two new series with no APE_label values. Thus, controlling the data flow with these tools might only be possible when applying them last or defining the tool sequence directly via constraints.
- **Tabular Data Manipulation:** Most tools of this category have two versions to produce a copy of the data with no label or keep the label and modify the data in-place. Both options are standard in typical data science workflow.
- **Constructors:** Nontrained models are part of the input for training, tuning, or ensembling. To avoid hardcoding their instantiation into these tools and, thus, further enlarging the tool taxonomy, constructors for these estimators are added to the toolset. Their inputs are either empty or contain a string variable to allow the user to pass on a list of estimators.

Another set of data types that need constructors is Matplotlib figures and axes. While most modifications are made after the tool produces the plot, setting the size and layout occurs during the figure and axis instantiation.

5.2.2 Type Taxonomy

Next, we will discuss the changes in the ontology's type taxonomy to reduce the search complexity. As shown in ??, the number of clauses is influenced by multiple factors: the workflow length, number of inputs, and number of type dimensions. Even in its most minor configuration, APE ran out of heap space after ten steps. For this reason, data science applications will have to be split into multiple independent sections, and the notebooks will be joined outside of APE. The number of inputs is outside the scope of APE and entirely depends on which data importance assumptions the user has already made before synthesizing the workflow. However, as was shown, it is possible to use many inputs to create short tool sequences to help make these assumptions and filter the data for the actual workflow.

Lastly, we will give a brief rundown of the changes concerning the type taxonomies' dimensions:

- The **DataClass** dimension is the primary data type and is vital for differentiating type states. It is comparable to the data type of the objects in the Python backend. Hence, the dimension is kept in the taxonomy.
- **StatisticalRelevance** is mainly for user convenience and clear separation of independent and dependent variables. The workflow might run without the dimensions, but the result could be misleading or incorrect. The dimension stays part of the ontology to avoid data leakage.
- The last two dimensions, **DataState** and **DataSetIndex**, are relevant for the executability of the workflow but add only little semantic value. Hence, they will be removed to reduce the complexity and allow workflow lengths of up to eight steps.

These changes to the ontology compromise the semantic quality of the synthesized workflows and their executability. They solve some of the challenges mentioned previously but also raise new concerns, mainly how tools with no output or no visible change in their parameter types influence the supposedly linear tool sequences produced by APE.

5.3 Auxiliary Tools

At this point, APE is able to produce tool sequences representing valid data science workflows. However, some auxiliary tools are still required to enable an automated workflow. The following subsections will introduce four new steps or tools in the APE workflow for data science.

5.3.1 Input Data Preparation

The tool is implemented as a Python script designed to load input data consistent with how it will be read in the subsequent APE-generated workflow. Upon reading the data frames, the tool detects the data type of each column and assigns a corresponding type state. The read feature labels are used as values for the **APE_label** dimension. By passing a list of dependent variables for each table to the tool, users enable it to annotate the columns' type states in the **StatisticalRelevance** dimension. If desired, the tool can split the table into separate data frames, each containing dependent or independent variable columns, and create respective type states for them.

PassengerID	Name	Sex	Age	Survived
int64	str	str	int64	int64
...

Table 5.1: Sample Schema Excerpt from the Titanic Data Set.

The sample ?? has three integer and two string-type columns and is annotated with its Numpy data types. Additionally, all columns are independent except for Survived. First, the tool matches these with their **DataClass** dimension counterparts and annotates the type states accordingly. Even though each column is identified by a string, the **APE_label** value, and is processed as such in the workflow to

notebook transformation, the ontology has a specific sub-taxonomy to mark data series that are part of a table to handle table-column associations. The input configuration, representing the table in APE, is shown in ?? and contains another entry for the table in addition to the five columns.

Listing 5.2: Input Config for Sample Table.

```
"inputs": [
  {
    "DataClass": ["IntColumn"],
    "StatisticalRelevance": ["IndependentVariable"],
    "APE_label": ["PassengerID"]
  }, {
    ...
  }, {
    "DataClass": ["IntColumn"],
    "StatisticalRelevance": ["DependentVariable"],
    "APE_label": ["Survived"]
  }, {
    "DataClass": ["MixedDataFrame"],
    "StatisticalRelevance": ["None"],
    "APE_label": ["titanic_train"]
  }
]
```

5.3.2 Output Parser

After APE finishes synthesizing valid tool sequences, additional tools are required to transform these into executable programs. Internally, APE represents each found solution with a graph encoded as a set of tool-, type-, and bind-predicates [kasalica2022synthesis]. Notably, all these graphs are stored in a single solution file. The output parsing process loads this file and transforms each of the contained declarative workflow definitions into a linear sequence of tools and their input parameter dependencies. A crucial part of this step is the extraction of the terminal tool and type predicates from among all their parent taxonomy terms produced by APE's encodings, followed by the reconstruction of type states. The latter is done by collecting and grouping the dimension values for each step and parameter index.

In specific scenarios, APE introduces a new plain type to ensure that all concrete data types are leaves within the type taxonomy. This must be accounted for when constructing the script with actual Python object types. Finally, the intermediary workflow format, and the last artifact before the notebook construction, is a JSON file containing the solution's inputs, outputs, and tool sequence. Each step object includes its tool, parameter type states, and which previous tool output each of its inputs is bound to, as shown in ??.

Listing 5.3: Step in JSON Workflow Encoding.

```
{
  "tool": "lineplot_12",
  "input": [
```



```

    {
      "DataClass": "IntSeries",
      "StatisticalRelevance": "IndependentVariable",
      "APE_label": ["Pclass"],
      "src": [0, 2]
    },
    ...
  ],
  "output": [
    {
      "DataClass": "Axes",
      "StatisticalRelevance": "NoRelevance"
    }
  ]
}

```

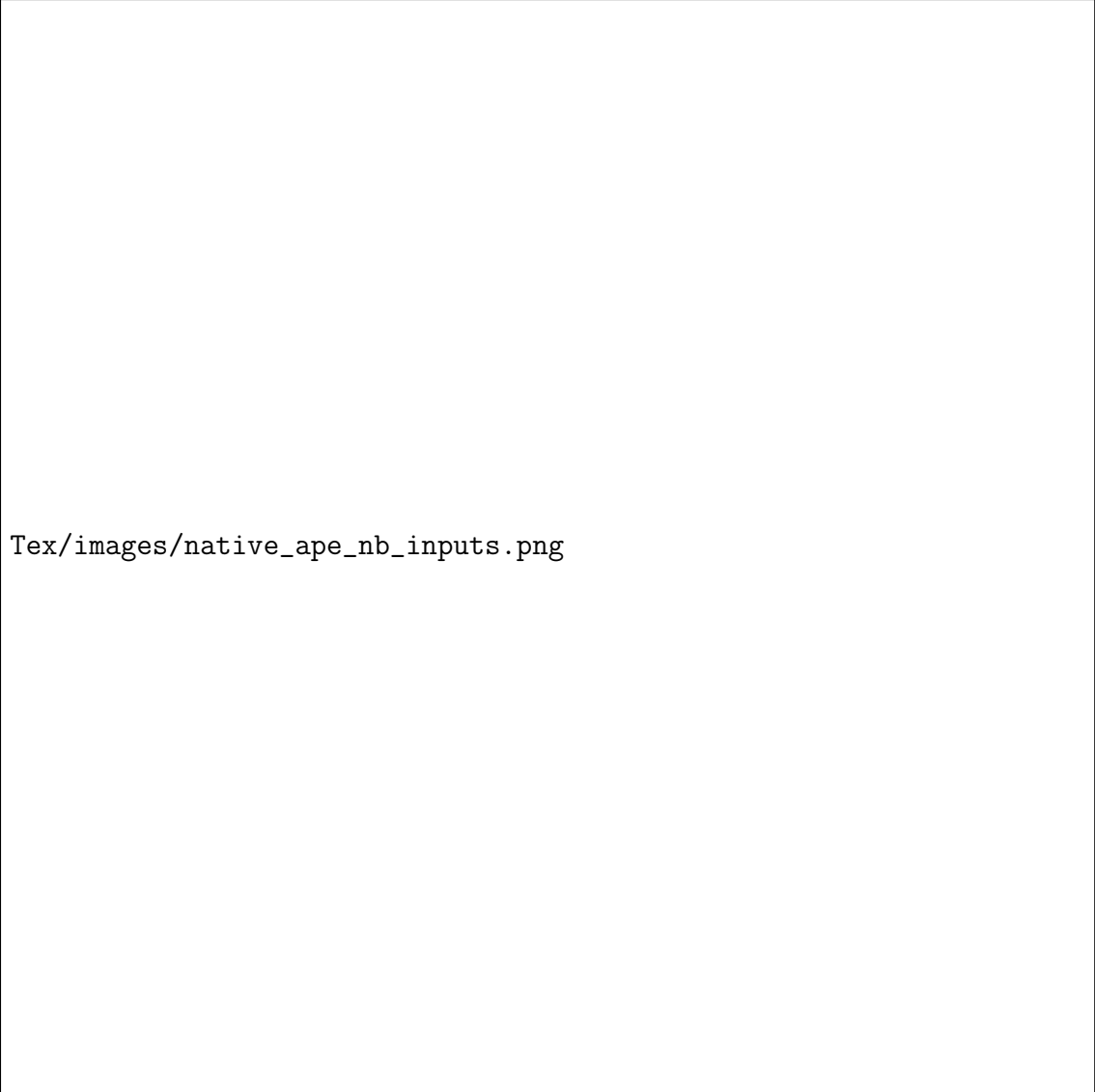
5.3.3 Python Tool Library

The tools in the data science ontology and their modes are implemented with Python and can be imported from this module. Each function implements exactly one tool, and multiple modes are represented by overloading the function signatures and adding the required branches to the function. The implementations bundle common function call sequences to simplify the interpretation and extension of the workflow notebooks, such that each step in the notebook is reduced to one line of code. Furthermore, they also include methods to display outputs and intermediary results, primarily tables, plots, and numeric statistics, where deemed sensible. While not necessarily part of the final workflow output, these tool outputs may interest the user and would otherwise disappear since they are immediately assigned to variables in the notebook.

In addition to the main libraries, which split the tools at the top level, the following libraries provide further functionality in the implementations:

- **Pandas** and **Numpy** to store and transform tabular data.
- **Scipy** for computing and checking statistical assumptions.
- **scikit-learn** for simple modeling.
- **Matplotlib** and **Seaborn** for creating graphs.
- **Gensim**, **Wordcloud**, **Spacy**, **BS4**, and **Textblob** for text analysis.

The function names and signatures are kept close to their source library equivalent to simplify extending and adapting the result for users that already have Python data science skills. Many tools are simplified by default. Optional keyword arguments allow the user to pass in more parameters outside of the APE workflow and, thus, will enable the customization of the results without increasing the synthesis complexity.



Tex/images/native_ape_nb_inputs.png

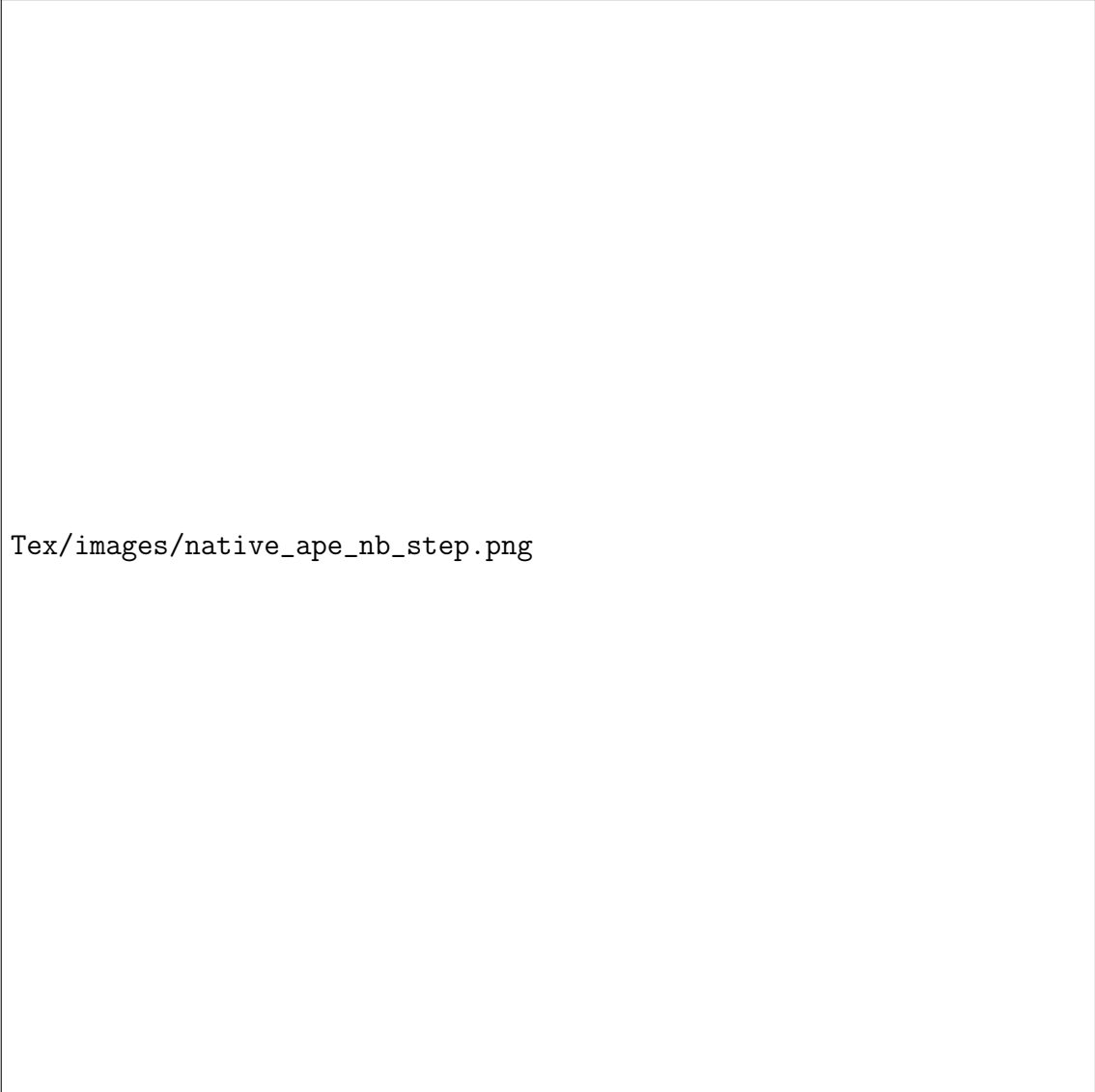
Figure 5.2: Workflow Inputs Section in Jupyter Notebook.

5.3.4 Executable Notebook Construction

After parsing them into a JSON file, each of the found solutions is finally transformed into a Jupyter Notebook. It provides the required framework to execute the script and displays the underlying code, markdown texts, LaTeX snippets, and images.

Header At the top of each notebook, the solution’s graph generated by APE is displayed for a general workflow overview. The necessary imports, such as the tool module and input loading libraries, follow this.

Inputs The next segment of the notebook, shown in ??, focuses on the workflow inputs. Each input is isolated in its own cell and annotated with its corresponding data state, identifier, and source in markdown. The symbol names are chosen by the user and defined during the data preparation.



Tex/images/native_ape_nb_step.png

Figure 5.3: Step Description, Code Cell, and Part of the Graph Output.

Tool Sequence Subsequently, the workflow steps are listed in the notebook’s primary section. Each step is structured to include a title with its step number and tool name, the type states for inputs and outputs, the dependency of each input parameter, the docstring for the tool being used, and finally, the function call, see ??.

The Python tool docstrings are listed under **Notes** and may elaborate on its functionality and arguments or include warnings about its usage. This is especially beneficial when the parameters APE assigned may not match the implementation’s semantic requirements, such as in the following scenarios.

- **Plotting:** The **hue** or **style** parameter may hold the label of a column with arbitrary type. However, too many unique values in that column, like in the case of a numeric identifier feature, might render the result less meaningful. For this reason, the example in ?? includes a warning concerning the hue parameter usage. There, the affected argument is matched with the **SalePrice** column of type float.

- **User-defined values:** Labels may be introduced in the input config. Since these potentially new types can not be checked in the tool transitions, they are passed directly through the notebooks and Python tools. The advantage of this is the simplification of the taxonomy while still allowing complex inputs. The disadvantage is that the unchecked input may be syntactically wrong or make no sense in the context.

A Python code cell contains the function call with its output assignment. The identifiers for these new variables consist of the data class, step number, and output parameter index for more straightforward interpretability. Input parameters are passed to the function with keyword arguments to remove any ambiguity. However, overloaded tools complicate pairing the type states to their correct Python keyword argument. Specifically, we can not differentiate multiple identical type states; thus, the first matching parameter is assigned by default. For example, plotting functions may use the optional parameters `hue` and `style` to adjust the look of the plot. However, both expect the same data type: It is unclear which keyword the third parameter in `??` is meant for. Since `hue` is listed before `style` in the function signatures, it is currently impossible to automatically assign the latter without assigning the first. Basic object and column type states, such as inputs 2 and 3, are passed to the function directly. In contrast, other complex types are passed as references to their previously created variables.

Outputs The final section of the notebook displays the workflow outputs. Similarly to the inputs and tool step sections, this includes their type states, the tool responsible for generating them, and their respective output indices.

5.4 Alternative ASP Backend

As used in this project, APE uses an off-the-shelf SAT solver [kasalica2022synthesis] to search for valid workflows, and the domain ontology and constraints are translated from more practical formats, such as Owl or JSON. The only other way to interact with the synthesis process is the composition of custom constraints using SLTLx. However, since even this expressive language is encoded into the SAT instance, many of the related challenges regarding user interaction remain. This section will introduce an alternative solver backend for the APE synthesis concept using ASP.

5.4.1 APE Encodings

The domain and workflow encoding are directly translated versions of their SAT counterparts. Where necessary, the predicates are split into multiple rules and constraints. `??` shows the set of ASP rules encoding the workflow from `??`. Specifically, this snippet represents the top-level generation of predicates with choice rules that will be filtered by the constraints. Similar to APE increasing the workflow length by one each time the solver does not find sufficient solutions, the ASP backend uses incremental solving with Python to iteratively instantiate and search. Here, for each new step `t`, the instance is extended by choices for the used tool, parameters, and data bindings: Lines 2 and 3 ensure that at least one tool is used during the

step. Lines 4 through 11 assign at least one type to each parameter position and dimension. Finally, lines 12 through 23 connect non-empty tool inputs to previously generated outputs from steps 0 to $t-1$ for each input index, with the workflow output being encoded as the input of step -1 . The empty type `eps` is the only value of the null dimensions and ensures exclusivity to other dimensions.

Listing 5.4: SAT Workflow Predicate Encoded as ASP Rules.

```

1 % [[W]]_n
2 1 = { use_tool(Tool, t) : term_tool(Tool) }.
3 1 { use_tool(Tool, t) : tool(Tool) }.
4 1 {
5     in((t, Ix), Dim, Type) : type(Dim, Type)
6 } :- Ix=1..MaxIn,
7     tool_input_ix_max(MaxIn).
8 1 {
9     out((t, Ix), Dim, Type) : type(Dim, Type)
10 } :- Ix=1..MaxOut,
11     tool_output_ix_max(MaxOut).
12 1 {
13     bind((t, IxIn), (0..t-1, 1..MaxOut))
14 } :- IxIn=1..MaxIn,
15     tool_input_ix_max(MaxIn),
16     tool_output_ix_max(MaxOut),
17     not in((t, IxIn), null, eps).
18 {
19     bind((-1, IxIn), (t, 1..MaxOut))
20 } :- IxIn=1..MaxIn,
21     tool_input_ix_max(MaxIn),
22     tool_output_ix_max(MaxOut),
23     not in((-1, IxIn), null, eps).

```

5.4.2 APE Extensions

Deploying an ASP backend instead of SAT provides the user with several new ways of interacting with the solver. The most notable are heuristics and soft constraints. Both enable the user to introduce domain and workflow knowledge in the form of preferences for the aforementioned choice rules. These constructs, exemplified in lines 1 through 8 of ??, change the search order to, e.g., favor plotting tools over modeling tools during the solving process or reduce the chances of an input appearing twice in a step. These preferences could improve the quality of the found results by ordering the vast amount of types and tool modes originating from general-purpose libraries. Moreover, in addition to natural language templates and SLTLx formulas, constraints could be encoded into the ontology directly while still being interpretable. For example, lines 10 through 15 show a hard constraint limiting visualization steps binding of `Figure` and `Axes` inputs to the same source step to ensure tool executability.

Listing 5.5: ASP Heuristics Encoding Domain and User Preferences.

```

1 % Usually no modeling

```

```

2  #heuristic in((t _), "DataClass", "SklearnObject").
3      [40, false]
4  #heuristic use_tool(t, "Modeling"). [40, false]
5  % Avoid using the same input twice
6  #heuristic in((t, X), "APE_label", COL):
7      in((t, Y), "APE_label", COL),
8      X != Y. [10, false]
9
10 % Use same source for figure and axes inputs
11 :- in((t, IxF), "DataClass", "Figure"),
12     in((t, IxA), "DataClass", "Axes"),
13     bind((t, IxA), (T1, _)),
14     bind((t, IxF), (T2, _)),
15     T1 != T2.

```

5.4.3 Future Work

While this alternative backend might improve on some usability aspects of APE, it also introduces longer solving times on larger workflow problems. The computational overhead from directly translated SAT encodings could be reduced by rewriting the rules from the ground up following ASP best practices. Notably, the most significant modification would be compromising on the general purpose ontology compatibility and tailoring the backend to the data science domain. As a result, the framework could include ASP rules for tool annotations with real generics and passthrough parameters, nested types, and weight optimization - mostly ASP native features.

Chapter 6

Evaluation

This chapter will delve into the applicability and efficiency of APE within the data science domain. It aims to assess whether APE can effectively generate workflows for the three selected use cases - EDA, Predictive Modeling, and Text Analysis - with a particular focus on user interactions like extension, customization, or repair of found solutions. To prevent out-of-memory errors, the use cases will be divided into multiple syntactically independent sequences. Next, the alternative ASP backend will be applied to selected tasks where potential advantages or disadvantages compared to the native APE implementation could be found. Finally, given the rising popularity of generative AI in code generation, the chapter will conclude with a brief overview of the experiments on workflow construction in the data science domain with ChatGPT [ChatGPT], Bard [Bard], and Copilot [openAI2021codex]. All experiment artifacts, such as generated notebooks, their scripts, or chat histories, can be found in the repository [PanRepo].

6.1 Native APE

This section will apply the APE framework to each of the three use cases: EDA, Predictive Modeling, and Text Analysis. Additionally, APE is configured to generate up to five solutions to examine the exploration potential. The evaluation focuses not on the generated notebooks' produced content but on the user interactions with APE and the data science ontology. For this reason, the use cases' Kaggle templates are simplified by only keeping unique tool sequences that might produce technically and semantically interesting workflows - Visualizing one numeric variable will not differ in data dependencies from visualizing another numeric variable, even if the use case would usually demand it. Each of the three workflows is split into multiple constraint sets whose synthesis results are evaluated separately. This is followed by a more interpretative discussion of the discovered benefits and challenges, see ??, of applying APE in each use case.

6.1.1 Exploratory Data Analysis

Most operations in this use case draw upon the unaltered input dataset. Hence, dependencies between steps are scarce. Furthermore, in the context of EDA, there is no prioritization of specific outputs; any generated plot or statistic could be considered a valid part of the EDA artifacts, eliminating potential output data constraints or

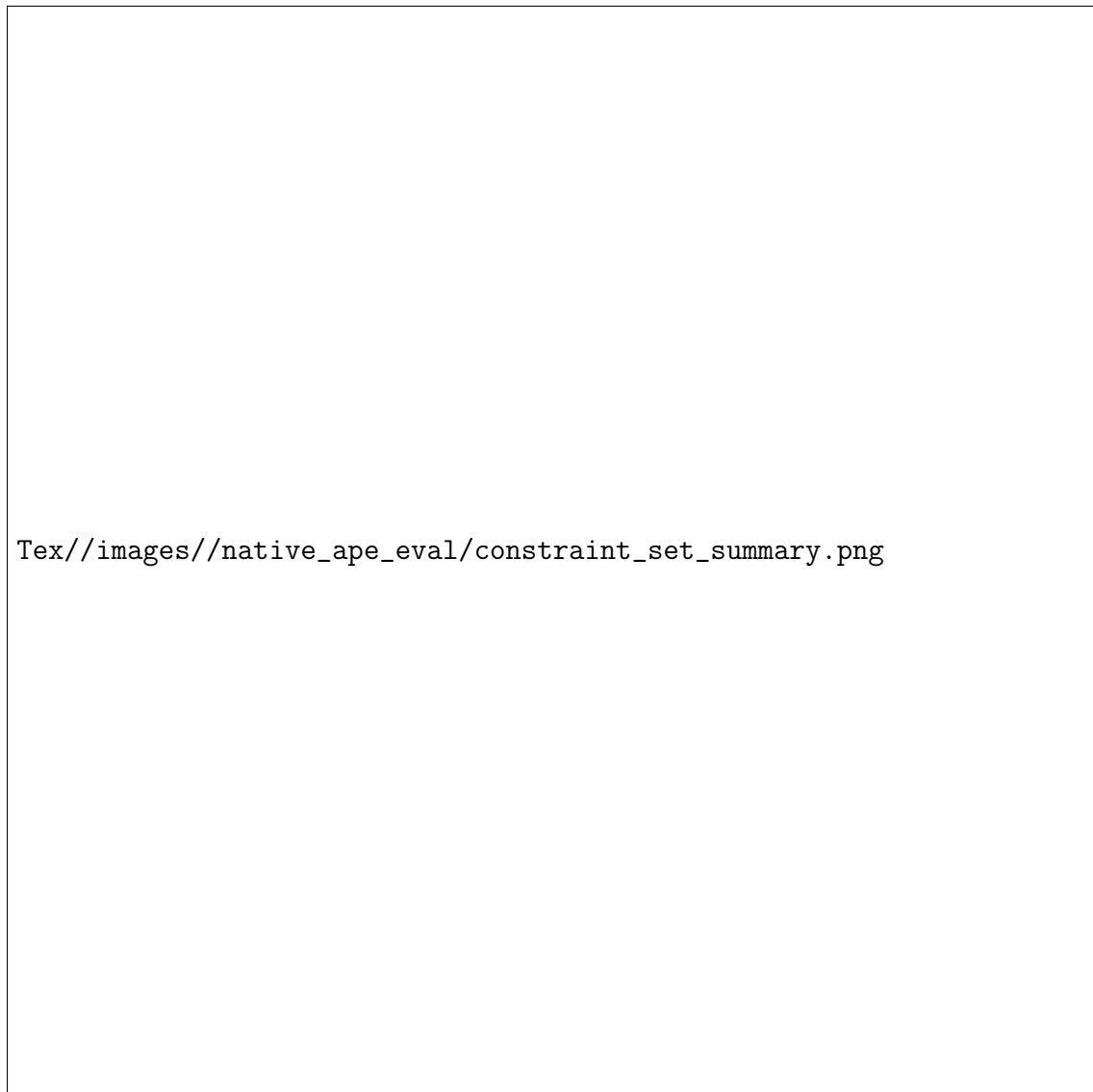


Figure 6.1: Overview of all Generated Notebook Cells across the Constraint Sets of the Three Use Cases.

dependencies. On rare occasions where tool sequences additionally transform data, prepending or appending a descriptive tool by duplicating the chosen step will effectively capture the changes. This relative lack of inter-tool dependencies enables the workflow segmentation into the four disjunct parts introduced in ??: 1. Exploring the Distribution of Univariate Dependent Variables, 2. Exploring Multivariate and Independent Variable Distributions, 3. Data Cleaning, and 4. Testing Statistical Assumptions.

Exploring Univariate Dependent Variable Distribution

The initial tool sequence exploring univariate distributions should compute three descriptive statistics or summaries and plot the dependent variable. Notably, no data must be passed between tools, so the constraint set is relatively simple. ?? contains the desired results and their groups of inferred APE constraints represented by their natural language templates.

Intention	Constraint
Describe the input data.	Use module <code>describe</code> .
Plot a distribution graph of the dependent variable.	Use <code>Distribution</code> with an input of type <code>DependentVariable</code> .
Calculate the skew and kurtosis of the dependent variable.	Use <code>skew</code> with an input of type <code>DependentVariable</code> .
	Use <code>kurt</code> with an input of type <code>DependentVariable</code> .

Table 6.1: Univariate Distribution Exploration Constraints.

The tools `skew` and `kurt` are defined with only two parameters: the source table and the column to calculate the statistic. Since only one table is available, supplying the type constraint for the single dependent variable column fixes the operation entirely, and hence, only one variation can be found in the solutions. Similarly, `describe` also uses the table and column parameters. However, the column parameter is optional: If used, the description is limited to the specified column; if not, all numeric or non-numeric columns in the table are described. Two of the five generated workflows outline the entire data set, while the others limit themselves to a single column.

Next, the `Distribution` module contains various plotting tools to visualize univariate or multivariate distributions. Furthermore, many have modes accepting parameters for additional customizations like `hue` and `style`. These degrees of freedom enable exploring different approaches to plotting the dependent variable distribution with APE. The results include:

- Two box plots with the column `SalePrice` across the y-axis, see ??. It is simple and gives an overview of quantiles and potential outliers. Furthermore, the plot shows the skewness in the `SalePrice` column.
- A joint plot with the `GarageArea` on the x-axis and `SalePrice` on the y-axis, as shown in ??. The graph shows the required distribution of the dependent variable, once via scatter and another time with a histogram. Also, the numeric independent variable `GarageArea` is included, showing a seemingly linear relationship between the two columns.



Figure 6.2: Generated Box Plot Showing the **SalePrice** Distribution.

- Another joint plot where **SalePrice** is plotted against itself. The histograms on the side still show the requested **SalePrice** distribution. Nonetheless, the scatter plot in the middle of the figure is arguably not adding any value.
- A histogram with inputs **GrLivArea** and **SalePrice**, as partially shown in ???. Here, the distribution is plotted for the ground floor living area across the different values of the dependent variable. Since **SalePrice** is also numeric, the number of unique values in its column is large enough to increase the height of the legend and figure multiple times and make the supposedly discrete hues indistinguishable. While the histograms barely resemble the actual **GrLivArea** distribution, they still show the increasing sale price with increasing ground floor living area values. Additionally, the figure still provides a rough idea of the distribution of the dependent variable and even shows two potential outlier bins.

Exploring Multivariate and Independent Variable Distributions

Including an independent variable enables the exploration aspect of APE by allowing the synthesis process to choose from various columns, thus providing semantically different outcomes. The desired results include:

- Plot a relationship and a distribution graph between the dependent and an independent variable. Adjust the figure size and label rotation to the high number of axis ticks.
- Plot a correlation heatmap and pair plot of ten features that correlate the most with the dependent variable.

To include arbitrary integer inputs in the process, such as figure size and feature count, additional `APE_label` inputs are temporarily added to the workflow configuration as outlined in ???. The extensive constraint set restricting the distribution graph and heatmap tool sequences can be seen in ??.



Figure 6.3: Unintended EDA Results.

Two of the five generated workflows, solutions three and five, contain faulty code and abort their executions. These errors occur during the `heatmap` step, the subsequent `rotate_x_labels` step that tries to use the `heatmap` results, and during the `pairplot` step.

The `heatmap` tool includes an optional pivot step, executed if the pivot columns are passed to the tool. These columns must exist in the table. However, since we removed the `DataSetIndex` dimension from the type taxonomy to reduce the overall complexity, APE created workflows trying to access columns that were dropped when the `k_most_corr_indep_var_corr_matrix` tool generated the correlation matrix. Even though other solutions use existing columns and, thus, generate syntactically valid workflows, the resulting notebooks do not present any relevant information: The `heatmap` tool is not intended to allow pivoting on correlation matrices, see ???. The other workflows contain the implicitly intended tool mode of the `heatmap` module, as shown in ???.

Next, the `pairplot` module is supposed to visualize the relation of various features to each other by generating a grid of scatter plots for each column combination. Optionally, the user may provide the identifier of the dependent variable and an integer number to display only the scatter plots of the n -most correlating features. The constraints enforce at least one parameter to be assigned to the `SalePrice` column. However, in one case, the column is mapped to the `hue` parameter, and the notebook instead displays n features that correlate the most to the number of garages. While the scatter plots still show the requested information, the selection and order of columns seem irrelevant to the `SalePrice` EDA process. The last solution aborts the tool step because the `—verb—hue—` is assigned to a column not available in the correlation matrix.

Another unintended result stems from the adjustment of the distribution plot. The targeted tool sequence encoded in the constraints is:

1. `set_figure_size` creates a new figure and axes pair.

Intention	Constraint
Plot a scatter relationship graph between the dependent and an independent variable.	Use <code>scatterplot</code> with an input of type <code>DependentVariable</code> .
	Use <code>scatterplot</code> with an input of type <code>IndependentVariable</code> .
Plot a distribution graph between the dependent and an independent variable. Also, adjust the figure size and label rotation to the high number of axis ticks.	Use <code>set_figure_size</code> with an input of type 16.
	Use <code>set_figure_size</code> with an input of type 9.
	<code>set_figure_size</code> should generate an output used by <code>Distribution</code> .
	Use <code>Distribution</code> with an input of type <code>DependentVariable</code> .
	Use <code>Distribution</code> with an input of type (<code>StrColumn</code> , <code>IndependentVariable</code>).
	<code>Distribution</code> should generate an output used by <code>rotate_x_labels</code> .
Plot a correlation heatmap of ten features that correlate the most with the dependent variable.	Use <code>k_most_corr_indep_var_corr_matrix</code> with an input of type <code>DependentVariable</code> .
	Use <code>k_most_corr_indep_var_corr_matrix</code> with an input of type 10.
	<code>k_most_corr_indep_var_corr_matrix</code> should generate an output used by <code>heatmap</code> .
Plot a pair plot of ten features that correlate the most with the dependent variable.	Use <code>pairplot</code> with an input of type <code>DependentVariable</code> .
	Use <code>pairplot</code> with an input of type 10.

Table 6.2: Multivariate Distribution Exploration Constraints.

2. `Distribution` uses the output pair and creates a resized plot.
3. `rotate_x_labels` uses the new output pair to rotate the x labels on that graph.

The constraints imply using at least one input from `set_figure_size` in `Distribution`. As a result, some notebooks use the generated axes, and some use the figure. In instances where there are previous plotting tools, APE may pass on one of the other outputs instead. Notably, as the last tool in the sequence, `rotate_x_labels` does not receive a complete plotting pair in any of the five solutions generated.

Finally, the scatter plot tool has even more optional parameters and, hence, more unintended results in the synthesized workflows. Four notebooks contain the `SalePrice` column in the y-Axis and assign various columns to the `hue` parameter. The remaining notebook plots the building year against itself and colors the markers according to the sale price, as shown in ?? . Furthermore, it uses the correlation matrix as the source table, removing any recognizable statistical relevance from the graph. Another solution shown in ?? contains the `SalePrice` variable as the `style` parameter, leading to Seaborn running out of distinguishable markers at a value of around 100,000.

Data Cleaning

This part of the EDA use case focuses on handling missing data and outliers. As before, the tool sequence will be a simplified representation of the complete workflow that can be adapted and extended by duplicating steps and exchanging column labels. The workflow should use four tools: `na_count_percentage`, `drop_na_col_i`,



(a) Heatmap Generated with Pivoted Correlation Matrix.



(b) Intended Heatmap of Correlation Matrix.

Figure 6.4: Heatmaps Generated by Different APE Solutions.

`filter_sd`, and `drop_sd_i`. Usually, the user would first inspect the data and then decide to drop specific data points. However, the user can reorder these four syntactically independent tools and, thus, simplify the constraint set:

Intention	Constraint
Calculate the percentage of missing data in each column.	Use module <code>na_count_percentage</code> .
Drop columns with missing data in-place.	Use module <code>dropna_col_i</code> .
Select a numeric feature and filter rows where the value might be an outlier.	Use module <code>filter_sd</code> .
Drop rows with potential outliers in-place.	Use module <code>drop_df_i</code> .

Table 6.3: Data Cleaning Constraints.

All generated notebooks are fully executable and identical in their tool parameter mappings except for the columns selected during the outlier exploration and elimination. The in-place modifications on the input table remove the tool dependencies, such that the sequences run in any order. Nevertheless, the results differ: Removing missing values will influence the display of missing values in a subsequent step. Reordering the tool steps fixes this semantic dependency. Notably, the outlier exploration relies predominantly on experimentation with various standard deviation thresholds. However, the notebooks provide preliminary views into potential outliers without requiring user-specified values, as this parameter is already set to a default value and omitted from the tool signature.

Testing Statistical Assumptions

The last APE run of this use case represents the testing of statistical assumptions. It synthesizes workflows, checking the normality of dependent and independent vari-



(a) Scatter Plot Showing a Variable Plotted against Itself.

(b) Another Scatter Plot Using the SalePrice as the style Parameter.

Figure 6.5: Heatmaps Generated by Different APE Solutions.

ables in our dataset before and after a transformation. As most numerical transformations in our toolset have the same syntax, we will only use `log` as an example in the constraints; see ??.

Intention	Constraint
Apply the log transformation to some column or the entire table.	Use module <code>log</code> .
Check the normality of the dependent variable.	Use <code>normality_plots</code> with an input of type (<code>Column</code> , <code>DependentVariable</code>).
Check the normality of an independent variable.	Use <code>normality_plots</code> with an input of type (<code>Column</code> , <code>IndependentVariable</code>).
At least one of the normality checks should use the log-transformed data.	<code>log</code> should generate an output used by <code>normality_plots</code> .

Table 6.4: Checking Normality Constraints.

As defined by the constraints, all five solutions have a length of three, with one of the normality plots showing the selected variable after its log transformation, as shown in ??. To examine the original and modified states for both dependent and independent variables, the constraints would have to be capable of distinguishing between multiple applications of the `normality_plots` function. Adding more constraints to fix the entire tool sequence would increase the search complexity and potentially be more time-intensive than manually duplicating the required tool cells and exchanging the column parameter.

Discussion

Most of the synthesized notebooks are executable in their entirety, and in the few cases where errors halt the run, the notebook can be manually resumed, skipping the affected cell. ?? overviews each partial workflow’s result across all five generated solutions. Only the notebooks for multivariate distribution exploration, which



Figure 6.6: A Normality Plot Using the Log-Transformed Data Produced in Step 1 of the Workflow.

Table 6.5: Summary of Code Cells across the EDA Constraint Sets.

Constraint Set	Steps	As Intended	Not as Intended, Maybe of Interest	Not as Intended, Semantic Error / Not of Interest	Not Executable
Univariate Distribution	20	14	4	2	0
Multivariate Distribution	35	22	1	9	3
Cleaning Data	20	20	0	0	0
Checking Normality	15	15	0	0	0

introduces data dependencies, are not always executable. The lack of constraints to order or directly map types to parameters leads to potentially interesting tool uses and additional dimensions in requested figures when many available columns match the parameters, as was the case in the first two APE configurations. Nevertheless, often, the information produced by these tools is presented ineffectively or unsatisfactory.

In cases of column switch-ups or incorrect use of parameters, the markdown tool notes, exemplified by `??`, enable the user to make these minor corrections. However, some tools, namely plotting tools with style options, may have many input and output parameters, creating large documentation cells interrupting the code and result displays. Nevertheless, the information provided in these cells is essential for modifying the generated notebooks. It may be possible to reformat the parameters into a table to make it easier to read.

If tools produce new data sets, APE often falsely assigns these tables instead of the input tables in subsequent steps. This incorrect use of different dataset types could be avoided with the `DataSetIndex` dimension. Alternatively, additional constraints to explicitly control the data flow would increase the proportion of usable results. Specifically, these constraints would require the given tool to receive an-



Figure 6.7: Tool Tip for `histplot` Hinting at Error in `??`.

other’s entire output.

Performing operations in-place, such as in the data cleaning solutions, increases the probability of notebooks executing successfully. This could give users the impression of a reliable workflow, even though it may disregard underlying semantic dependencies.

Most of the time, at least one notebook had the desired results for every request. Users could select these tool cells and stitch together a new notebook, fulfilling all their constraints while adding in the out-of-scope but still relevant results. Overall, the dependency and data flow sequences in this EDA use case are relatively short, resulting in any faulty steps having minimal impact on subsequent steps.

6.1.2 Predictive Modeling

The second use case trains a predictive model and thus includes a higher proportion of transformations and, hence, more dependencies than the EDA workflow. However, due to the available in-place data manipulation tools, we can split it into three partial sequences to lower the search complexity in APE and stitch the resulting notebooks back together without modifying the code cells: 1. Feature Engineering, 2. Model Training and Evaluation, 3. Ensembling and Evaluation.

Feature Engineering

This first partial workflow uses transformation and encoding tools to preprocess the titanic data set for use with sci-kit-learn’s models. The constraint set for this configuration is rather extensive and detailed, as can be seen in the excerpt in `??`.

All new string and integer inputs are encoded into new APE inputs. However, the search problem with nine columns from the input data set¹, four new inputs, and a minimum of nine steps is too complex, leading to a memory error during

¹During the hypothetical EDA stage, the user would have decided to drop the columns `PassengerID`, `Ticket`, and `Cabin`.

Intention	Constraints
Extract the title prefix from the Name column.	Use <code>extract_i</code> with an input of type Name . Use <code>extract_i</code> with an input of type <code>'([A-Za-z]+)\.'</code> .
Replace rare extracted titles in the Name column with the string <code>'Rare'</code> .	Use <code>replace_i</code> with an input of type Name . Use <code>replace_i</code> with an input of type <code>'Lady Countess Capt Col Don Dr Major Rev Sir Jonkheer Dona'</code> . Use <code>replace_i</code> with an input of type <code>'Rare'</code> . If <code>replace_i</code> is used, <code>extract_i</code> must have been used previously.
Map nominal Age values into 20 equal-sized bins.	Use <code>bin_nominal_i</code> with an input of type Age . Use <code>bin_nominal</code> with an input of type <code>'20'</code> .
Map nominal Fare values into quartiles.	Use <code>bin_nominal_q_i</code> with an input of type Fare . Use <code>bin_nominal_q_i</code> with an input of type <code>'4'</code> .
Impute missing values in the column Age with the median.	Use <code>imput_median_i</code> with an input of type Age .
One-hot-encode the column Name .	Use <code>one_hot_encode_i</code> with an input of type Name .
Encode columns only after the transformations on the underlying data are finished.	If <code>Encoding</code> is used, do not use <code>EDAFeatureEngineering</code> subsequently.

Table 6.6: Feature Engineering Constraints Excerpt.

the solving process. The adapted constraint set reduces the complexity for APE while requiring some manual changes in the notebooks by the user: The integer 4 temporarily replaces 20, and two of the three near-identical one-hot-encoding steps are dropped.

All notebooks run without errors after reversing these changes in the synthesized workflow. As with previous tool sequences, unless fixed with constraints, the ordering varies and may disrupt semantic dependencies. For instance, string modifications and encodings have explicitly specified orders and occur in the correct order in all notebooks. By contrast, some notebooks unintentionally handle missing values before binning. Finally, the regular expression replacement tool uses two string parameters, which are indistinguishable except for the `APE_label` dimension. Consequently, some notebooks contain steps where the search pattern substitutes the replacement string. This switch-up carries over into the subsequent one-hot-encoding steps and the final preprocessed table.

Model Training and Evaluation

The model training workflow is a well-defined sequence. Most degrees of freedom stem from the model selection or hyperparameter tuning, which are not part of this partial sequence. Since all tools implement non-in-place operations on data sets indistinguishable by APE, the data flow will be defined mainly through tool-connection constraints.

None of the resulting notebooks are free of errors. While the first three steps of every workflow, initialization of a classifier, shown in ??, and the train-test-split sequence run as intended, none of the fitting and evaluation calls are executable. Like the graph customization tools, these require multiple types to be passed between them:

- `train_test_split` generates four outputs: a training feature table, a test

Intention	Constraints
Split the input table into training and test feature and label sets.	<code>column_split</code> should generate an output used by <code>train_test_split</code> .
Train a classifier on the new training set.	<code>train_test_split</code> should generate an output used by <code>fit_estimator</code> . Use <code>fit_estimator</code> with an input of type <code>Classifier</code> .
Predict the labels of the test set.	<code>train_test_split</code> should generate an output used by <code>predict</code> .
Generate a classification report for the test prediction.	Use <code>classification_report</code> with an input of type <code>Prediction</code> . <code>train_test_split</code> should generate an output used by <code>classification_report</code> .

Table 6.7: Modeling and Evaluation Constraints.



Figure 6.8: Faulty Notebook Cell Fitting the Estimator with an Incompatible Matrix-Vector Pair.

feature table, a training label series, and a test label series.

- `fit_estimator` requires a table and series of equal length. However, without the dimension, `DataSetIndex` training and test sets are indistinguishable by APE, leading to potentially inconsistent numbers of samples. Furthermore, the tool-connection constraint only requires the `fit_estimator` tool to use one of the `train_test_split` outputs. If, as exemplified by ??, the data set split produced the training table, the label series might be from a previous tool.
- `predict` only uses one data set input and runs in all notebooks after fixing the other cells.
- `classification_report` has the same data set parameter types as the tool `fit_estimator`; thus, their cells have similar error sources.

Ensembling and Comparison

Ensembling enables the user to request an arbitrary number of models for aggregation. To address this obstacle related to explicit parameters in tool modes, as discussed in ??, the factory tool for basic ensemble models employs a single string list of model identifiers and combines their constructor calls as shown in ?. The user can alter the ensembled models by adjusting this string list following the tooltips provided. The APE configuration for training an ensemble model and comparing it to the baseline resembles that of basic models, with the sole addition including a string model list as an input. As a result, all five generated notebooks contain similar faulty `split-train-predict-evaluate` tool sequences.

Discussion

Table 6.8: Summary of Code Cells across the Predictive Modeling Constraint Sets.

Constraint Set	Steps	As Intended	Not as Intended, Maybe of Interest	Not as Intended, Semantic Error / Not of Interest	Not Executable
Feature Engineering	35	32	0	3	0
Modeling, Evaluation	30	20	0	0	10
Ensembling, Evaluation	30	20	0	0	10

Variable user-defined inputs work well with the `APE_labels`, as shown by the feature engineering configuration change in the first sequence and the classifier initialization steps in the last two, which use predefined strings with no source tool or user input strings. Notably, the implicit dimension encodes potentially arbitrary parameter values outside the data science ontology while still being documented by the markdown cells; see ?.

The rigid data flow structure in modeling workflows necessitates the user to compose a more detailed set of constraints than earlier APE configurations. Despite this effort, the received notebooks may contain invalid modeling solutions; see ?. Specifically, the feature engineering tool sequences use in-place transformations and, hence, are less vulnerable to unexpected parameter assignments. In contrast, training and evaluating a model generates separate data sets that might not be distinguishable with taxonomy terms. The extensive and technical nature of the applied constraints, even with the abstraction layer provided by the ontology, could make manual notebook coding a more practical and effective approach. Moreover, while varying sub-tools and column mappings proved beneficial in other tool sequences, such flexibility would undermine the statistical significance of the `split-train-predict-evaluate` sequence if parameters were swapped indiscriminately.

6.1.3 Text Analysis

Lastly, the IMBD text-based dataset is a test for assessing APEs and the data science ontology's proficiency in generating workflows that process complex data

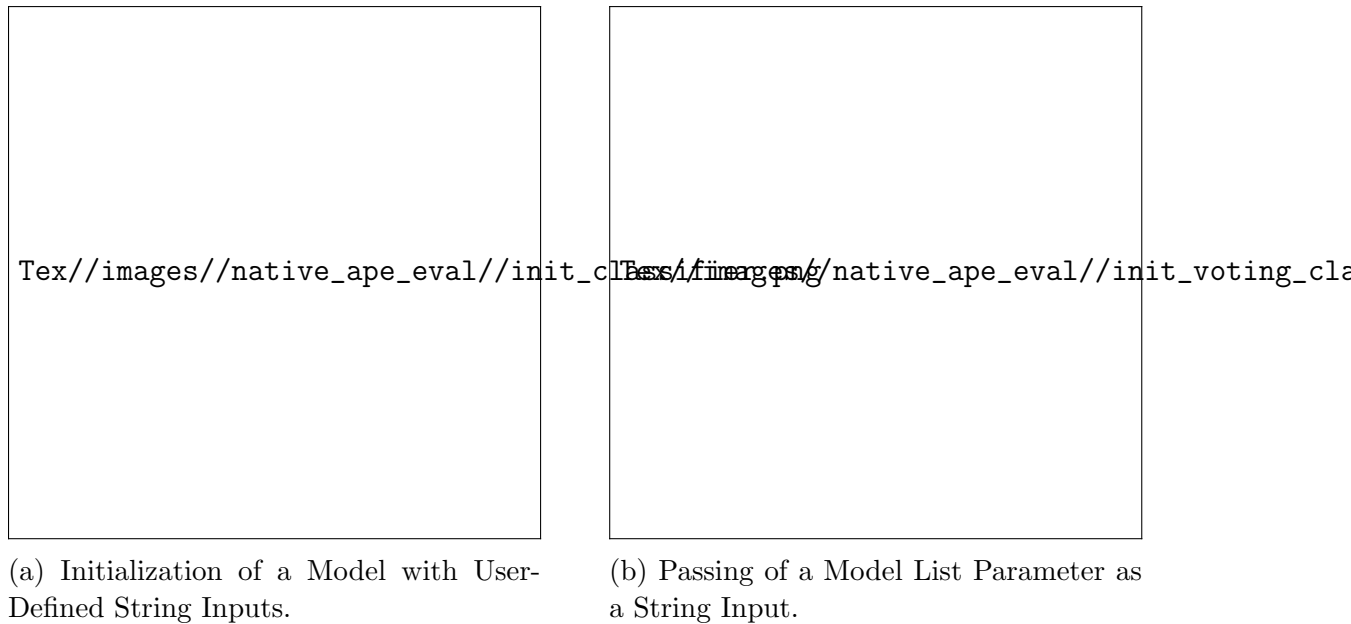


Figure 6.9: Model Initialization Code Cells with Detailed Markdown Annotations.

types opaque to the user and more challenging to interpret within the notebook than earlier data frame types. Following the Kaggle template, the unstructured review data undergoes preprocessing before model training. The workflow is split into two segments: 1. Text Preprocessing and 2. Embedding and Modeling.

Text Preprocessing

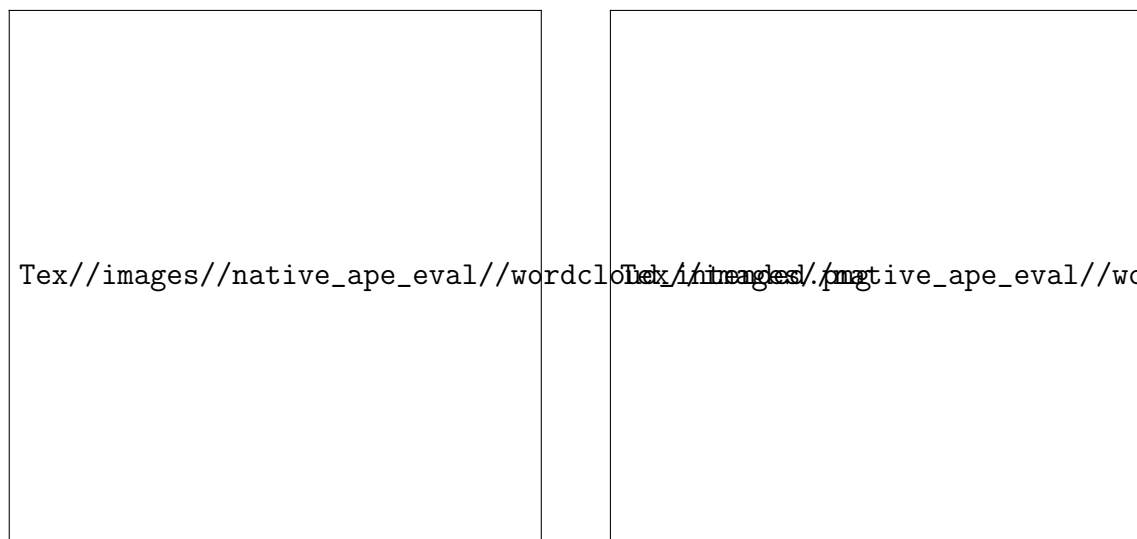
The generated tool sequence will modify the data for the later embedding and modeling steps. Since these workflows are developed separately, all input transformations must operate in-place to preserve the other workflows' data independence, similar to the feature engineering section of the previous use case. Additionally, the data set only has one feature column: the review texts. Thus, constraints ordering the steps must be supplied to comply with the tools' dependencies while modifying this variable.

Among the five produced solutions, only two have a length of six steps, the minimum for this constraint set. The subsequent three workflows incorporate an additional seemingly arbitrarily chosen tool, leading to one notebook with incorrect parameter mappings. Another prepends the `train_test_split` tool and occasionally references the wrong table for its operations. The final workflow among the longer ones also includes the `train_test_split` module but disregards the output in the following tool cells. Conversely, the two minimal notebooks are identical except for the arrangement of parameters in the one tool using two string parameters that APE cannot differentiate with the data science ontology: `replace_re_i`. The interchanged replacement patterns substitute whitespace characters with the intended search pattern. Nevertheless, the result remains interpretable, as seen in the comparison in ??, and correcting this manually is relatively straightforward. Either way, the time consumed by APE's runtime or by manual adjustments to the notebooks pales in comparison to the duration required to process the text in 50,000

Intention	Constraints
Extract text from HTML in the review column.	Use <code>get_text_from_html_i</code> with an input of type <code>review</code> .
Next, expand abbreviations in the review text.	If <code>get_text_from_html_i</code> is used, the next module in the sequence should be <code>expand_abbr_i</code> .
	Use <code>expand_abbr_i</code> with an input of type <code>'abbrev.json'</code> .
	Use <code>expand_abbr_i</code> with an input of type <code>review</code> .
Next, remove any non-alphabetical characters in the review text.	If <code>expand_abbr_i</code> is used, the next module in the sequence should be <code>replace_re_i</code> .
	Use <code>replace_re_i</code> with an input of type <code>review</code> .
	Use <code>replace_re_i</code> with an input of type <code>'[^a-zA-Z]'</code> .
	Use <code>replace_re_i</code> with an input of type <code>' '</code> .
Next, lemmatize the review text.	If <code>replace_re_i</code> is used, the next module in the sequence should be <code>lemmatize_i</code> .
	Use <code>lemmatize_i</code> with an input of type <code>review</code> .
Plot a word cloud from the review text as the last step.	Use <code>plot_wordcloud</code> as the last module in the solution.
	Use <code>plot_wordcloud</code> with an input of type <code>review</code> .

Table 6.9: Text Preprocessing Constraints Excerpt.

reviews².



(a) The Intended Review Text Word Cloud.

(b) The Word Cloud Generated with Swapped Substitution Patterns.

Figure 6.10: Wordclouds Generated by Different Solutions.

Embedding and Modeling

The modeling workflow for textual data is similar to the basic one in that it requires the `split-train-predict-evaluate` sequence. However, it further necessitates embedding the textual column into numeric matrices before the model can be trained. This embedding is fitted and applied to the training data and then reapplied to the

²The spelling correction step is exceptionally time intensive and runs for multiple hours on the Apple M1 Max chip.

test data for evaluation. These novel data flow requirements are added to an already fragile modeling data dependency set.

Intention	Constraints
Embed training review text using Word2Vec.	<code>train_test_split</code> should generate an output used by <code>embed_text_word2vec</code> .
	Use <code>embed_text_word2vec</code> with an input of type review.
Embed the test review data using the trained Word2Vec model.	Use <code>embed_text_word2vec</code> with an input of type Word2Vec.
	<code>embed_text_word2vec</code> must have used <code>train_test_split</code> prior to it.
Train a classifier using the embedded training data.	<code>embed_text_word2vec</code> should generate an output used by <code>fit_estimator</code> .
	Use <code>fit_estimator</code> with an input of type Classifier.
Predict the labels of embedded test set.	<code>embed_text_word2vec</code> should generate an output used by <code>predict</code> .
	<code>fit_estimator</code> should generate an output used by <code>predict</code> .

Table 6.10: Text Classification Constraints Excerpt.

This APE configuration generates suboptimal solutions similar to the earlier modeling and evaluation scenario. None of the notebooks are entirely executable; four of the five proposed workflows mismatch data inputs in the model fitting step and one in the evaluation step. The only tool sequences generated reliably have their order and input parameters fixed by constraints. Additionally, repairing the faulty code cells and further extending the notebook is more convoluted than in the previous modeling workflow. In the case of basic tables, users may directly inspect the schema and content of passed parameters to identify the issues. However, interpreting the embedded text matrices presents a difficult task to the user due to their structure. These matrices are often characterized by their large size, sparsity, high dimensionality, and lack of a direct, identifiable connection to their originating texts, as exemplified in ??.

Listing 6.1: Embedding Matrix of Review Texts.

```
array([[ -0.14810246,  0.21811058, ...,  -0.16859908,
         0.17599331,  0.24811591],
       [ -0.14908928,  0.21933018, ...,  -0.16867837,
         0.1750795 ,  0.24900335],
       ...,
       [ -0.15202075,  0.22132936, ...,  -0.1739105 ,
         0.17905356,  0.25420347]], dtype=float32)
```

Discussion

As expected, fixing each step in the feature engineering workflow narrows down the variety of sequences that APE can generate. Moreover, even though the chosen text data set only contains two columns, both are of type string and, thus, valid inputs for the applied preprocessing tools. Consequently, the workflow configuration must restrict the operation inputs to the review text column. As a result, the constraint

Table 6.11: Summary of Code Cells across the Text Analysis Constraint Sets.

Constraint Set	Steps	As Intended	Not as Intended, Maybe of Interest	Not as Intended, Semantic Error / Not of Interest	Not Executable
Text Preprocessing	33	26	0	6	1
Embedding, Modeling	40	25	1	1	13

set includes the used tools, their input parameters, and the execution order. In other terms, the entire workflow is already defined in such technical detail that the user may as well compose the notebook themselves.

Embedding textual data adds another step to the modeling workflow. Specifically, the required tool must be applied separately to training and testing data, forming even more potential combinations of parameter bindings in the APE workflow encoding, with only one being intended and statistically sound. While the introduction of the new data type `EmbeddingMatrix` reinforces the separation of different phases in the data flow in the same way the `DataSetIndex` dimension would have, the higher tool sequence complexity and more challenging modifications of generated notebooks may make the manual composition of such workflows more appealing.

6.2 Alternative ASP Backend

The ASP variant of APE produces similar results to the SAT version if run with no additional encodings. It performs better on smaller and fewer solutions since the instantiation of an ASP program already uses rules and facts to reduce the required resources, especially the setup time. No out-of-memory errors occurred while experimenting with different configurations. However, this trend reverses when moving to configurations searching for less restricted, longer, or more workflows. The non-optimized encoding seems suitable for the three use cases but slows down on, e.g., the EDAM APE workflows.

Introducing domain heuristics does not significantly influence the solver runtime³. By contrast, it resolves many of the problems previously appearing in the generated notebooks, with most of the constraint sets having at least one valid solution. For instance, the EDA use case proved to be a suitable candidate for exploring found solution alternatives with native APE. However, the additional style parameters and data dependencies while examining multivariate distributions lead to several mismatched bindings. The ASP backend improves that aspect by initially directing the search towards the more likely preferred solutions but eventually also finding the faulty ones.

Finally, the user may write workflow-specific heuristics and constraints to customize the synthesis process further. However, for most of the target audience, this could be comparable to manually extending the APE SAT encoding or using only SLTLx to compose constraints.

³Most of the tested configurations for five solutions finished in under 5s, with others taking up to 10s on the Apple M1 Max. Domain heuristics changed these measurements by less than 1s in either direction.

6.3 Generative AI

Language models such as GPT4 [openai2023gpt4], PaLM2 [anil2023palm], or Codex [openAI2021codex] enable conversations with users in natural language without restricting them to templates or simple grammars. Tuned variants of these models power the popular services OpenAI ChatGPT [ChatGPT], Google Bard [anil2023palm, Bard], and GitHub Copilot [openAI2021codex, copilot2023robustness]. This section will briefly explore the user interaction with these services, creating data science workflows. Notably, the focus will be on some unique key features of each service, such as using files as input or accessing the internet.

6.3.1 ChatGPT

The experiments with ChatGPT are centered around the ability to upload data and receive a semantic summary of the contained information. Therefore, the EDA use case with the housing data set is chosen as a starting point. After uploading the file, the model is prompted to perform several tasks: EDA, data cleaning, and checking statistical assumptions for modeling. The results follow the usual patterns and contain tools from typical Python libraries, such as Pandas, seaborn, and Scipy, with very little user input compared to the constraint sets required by APE. Furthermore, the model seems to present sufficient semantic understanding of the data to document and sometimes even explain the results in a human-readable form. This is especially useful with this data set due to the high number of columns the user would have to inspect manually. Occasionally, it will also outline the required steps to go deeper into a specific aspect and stop to prompt the user for further input. For instance, decisions regarding specific transformations to comply with statistical assumptions are left to the user.

Since this data set is relatively popular in the data science domain, another newer one with a different schema is uploaded to check the model's consistency and potential strict reliance on existing examples. The new table containing house prices from India⁴ includes fewer columns yet relatively similar content. When prompted with identical tasks, the generated workflow mostly follows the same structure and is accompanied by similar explanations of each step. Interestingly, the file name `IndianHouses` also serves as part of the context and refines the result documentation by, e.g., appending the Indian currency to prices.

ChatGPT provides the user with a conversational interface to their data and shows a shallow semantic understanding of it, enough to perform and document simple data science tasks, and most of the time, correctly; see ???. For instance, the graph in ?? was sketched by APE before further manual customization. However, the chats are session-limited, and while the model performs well on smaller classification workflows, such as the titanic use case or an employee churn data set⁵ it struggles with larger tables and more complex model as is the case with the IMBD movie review classification⁶. Moreover, the service's GPT4 version seems to be lim-

⁴The data set can be found on Kaggle: <https://www.kaggle.com/datasets/aemyjutt/indianhousesdataanalysis/>.

⁵Employee churn is a common classification target; however, the data set is newer than the knowledge base of the used ChatGPT version: <https://www.kaggle.com/datasets/tawfikmetwally/employee-dataset/data>

⁶The Advanced Data Analysis extension reduced the sample size before spelling correction to

You can see that for most of the samples, the model’s predicted probabilities align well with the actual outcomes. For instance:

- In the first row, the model predicted a 11.57% chance of the employee leaving, and the actual outcome was that the employee left (1).
- In the second row, the model predicted a 9.77% chance of the employee leaving, and the actual outcome was that the employee didn’t leave (0).

Quote 6.1: ChatGPT Mismatching Narrative and Displayed Data.

ited regarding its knowledge of more complex libraries or uncommon tool options. Finally, ChatGPT’s conversational nature could make it a useful addition to the APE workflow composition process to create and query constraint sets. Upon providing the list of APE’s constraint templates and the constraint table from ?? as input, the model successfully drafts a new set, shown in ??⁷, when prompted with a simple modeling task.

Table 6.12: Constraint Set for Training and Evaluating a Classifier Generated by ChatGPT.

Intention	Constraint
Load the dataset	Use <code>load_data</code> with an input of type <code>Table</code> .
Partition the data	Use <code>train_test_split</code> with an input of type <code>Table</code> . <code>train_test_split</code> generates <code>Train</code> and <code>Test</code> types. Outputs used by <code>train_model</code> and <code>evaluate_model</code> .
Train the model	Use <code>train_model</code> with an input of type <code>DependentVariable</code> . <code>train_model</code> generates <code>FittedClassifier</code> . Output used by <code>evaluate_model</code> and <code>save_model</code> .
Evaluate the model	Use <code>evaluate_model</code> with an input of type <code>DependentVariable</code> . <code>evaluate_model</code> generates <code>Prediction</code> .
Generate metrics	Use <code>performance_metrics</code> with an input of type <code>Prediction</code> . <code>evaluate_model</code> output used by <code>performance_metrics</code> .
Optimize parameters	Use <code>hyperparameter_optimization</code> with <code>FittedClassifier</code> . <code>train_model</code> output used by <code>hyperparameter_optimization</code> . Output used by <code>evaluate_model</code> .
Save the model	Use <code>save_model</code> with an input of type <code>FittedClassifier</code> .

6.3.2 Bard and Copilot

The examined version of Bard⁸ does not provide the option to upload data sets for analysis. It relies on information provided in prompts, its knowledge base, and its

100 entries and still timed out multiple times. Reducing it even further limits the statistical significance of all the following steps.

⁷This LaTeX representation of the constraints was also generated by ChatGPT.

⁸At the time of writing, Bard is still in an experimental phase.

access to the internet. While it claims to be able to load files from cloud storage, such as Google Drive, multiple attempts at experiments with the employee turnover data set were cut short consistently. On the few occasions where Bard seemed to be able to load the data from the provided link, identical prompts to access the file in the same chat had very different outcomes. Bard showed an extensive summary of columns, the contained information, and what research questions could be answered with it. However, the results were fabricated, and none of the data matched the actual original schema. The required context for this likely came from the prompt mentioning the data set's name. Additional attempts at trying to compose and execute the workflow with Bard relying only on data contained in prompts lead to more mixed results. On the one hand, it seems to understand the context of the provided modeling task and creates a suitable set of Python code snippets, all of which are accompanied by some explanation. On the other hand, trying to get Bard to execute these cells will sometimes lead to fabricated outputs between real results.

Copilot provides multiple options for users to interact with it. One is a chat, where the user can prompt it for code snippets, explanations, and improvements. Another option is the code-completion feature, where the user may start typing out comments, code, or docstrings, and Copilot will suggest likely continuations based on the current context and its knowledge base of common patterns. Particularly useful is the inclusion of the full code base from the open editor when suggesting completions. As a result, a developer may even use Copilot to generate code based on their own packages. Specifically, Copilot successfully suggested and extended partial workflows based on the library of data science tool wrappers used in this thesis while composing the baseline notebooks for each use case. However, the autocompletion will continue in whatever direction the user starts typing, even if it is semantically incorrect. As the recommendations are heavily influenced by the surrounding context, comments or other documentation are usually included in the generated output if they are present in the existing code base. All in all, Copilot may assist technically proficient users in composing data science workflows more efficiently. It is likely not a suitable tool for users with little to no programming experience since they may not be able to interpret the code recommendations and their quality.

Chapter 7

Conclusion

This thesis aims to answer the research question of whether the **ape!** (**ape!**) can be applied to workflows in the data science field. To this end, ?? first limits the scope of potential use cases to **eda!**, predictive modeling, and text analysis, and based on that, models the domain ontology. Next, ?? elaborates on challenges and solutions while integrating the ontology into the **ape!** synthesis process. Additionally, it presents an alternative solving backend based on **asp!** (**asp!**) that introduces heuristics and soft constraints to the workflow search. Finally, ?? evaluates the proposed approaches on the previously defined use cases and discusses how the lack of workflow-specific semantic information in the ontology affects them differently. It concludes with a comparison to the alternative backend and an overview of workflow construction experiments with generative **ai!**.

The defined ontology is modeled after the goal of producing general-purpose data science workflows using tools from standard Python libraries. However, this lack of semantic restrictions and the low-level nature of these operations result in the loss of type state dependencies in many tool sequences that seem essential for **ape!** to generate meaningful solutions. With exception to the **eda!** experiments, the produced Jupyter notebooks often contain non-executable steps or semantic errors, removing any statistical significance from the workflow artifacts. In contrast to the other domains **ape!** has been used in, the underlying types are not changing: Tables and columns are the data science ontology’s primary types, and only a few tools modify them in their transition rules, further removing data dependencies. This affects feature engineering, text preprocessing, and modeling sequences the most, and thus, their constraint sets must include nearly all parameter assignments and tool orders. Creating these requires a deep technical understanding of the underlying tools and types, at which point, the user may prefer to compose the notebooks manually and benefit from the additional flexibility. However, the **eda!** experiments show how **ape!** can synthesize data science workflows with short and independent tool sequences. Any degree of freedom usually stems from varying column and style choices, which users may explore in the various generated solutions.

These problems mostly persist in the **asp!** backend as it uses the same core concept. However, the addition of soft constraints encoding domain, workflow, and user-specific solution preferences reintroduces some of the lacking semantic context and, hence, can improve the quality of the notebooks significantly without changing the search space. Furthermore, **asp!** enables the user to compose hard constraints directly interacting with the problem encodings.

The presented results are limited by the lack of specific application domains. Future versions of data science ontologies could increase contained semantic context by introducing a new dimension modeling the various states of workflows from the target domain. To counter the increased search complexity, a new version of **ape!** using **asp!** would tailor its encodings to the data science field by, e.g., including core concepts, such as tables and columns, in the workflow definition. Experiments with generative **ai!** have shown the potential for conversational interactions of users with constraint sets lowering the skill threshold for customizing **asp!** constraints without the limitations of templates.

This thesis lays the foundation for a prospective novel approach to automated construction of data science workflows with **ape!** and **asp!**. While challenges remain, the addition of **asp!** shows a promising path to improve the workflow quality and user experience. As generative **ai!** matures, it may significantly enhance semantic modeling and constraint composition, enabling the synthesis of more complex workflows while lowering the technical barriers for non-expert users.