
**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS**

**SCHOOL OF SCIENCES**
**DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

**”DATA SCIENCE AND INFORMATION TECHNOLOGIES (DSIT)”**
**SPECIALIZATION: BIOINFORMATICS - BIOMEDICAL DATA SCIENCE**


**MSc THESIS**

# Dataflow Representation, Model Inspection and Explainability in ML Pipelines


**Stefania Patsou**

**Supervisors: Theodoros Dalamagas**, Research Director ATHENA RC


**ATHENS**

**DECEMBER 2024**


**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**”ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΕΣ ΠΛΗΡΟΦΟΡΙΚΗΣ (DSIT)”**
**ΕΞΕΙΔΙΚΕΥΣΗ: ΒΙΟΠΛΗΡΟΦΟΡΙΚΗ - ΒΙΟΙΑΤΡΙΚΗ ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ**


**ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ**

# Αναπαράσταση Ροής Δεδομένων, Επιθεώρηση και Εξηγησιμότητα Μοντέλων σε Ροές Εργασίας Μηχανικής Μάθησης


**Στεφανία Πάτσου**

**Επιβλέποντες: Θεόδωρος Δαλαμάγκας**, Διευθυντής Ερευνών ΕΚ ΑΘΗΝΑ



**ΑΘΗΝΑ**

**ΔΕΚΕΜΒΡΙΟΣ 2024**

**MSc THESIS**

Dataflow Representation, Model Inspection and Explainability in ML Pipelines

**Stefania Patsou**
**S.N.:** DS2190014

**SUPERVISORS: Theodoros Dalamagas**, Research Director ATHENA RC

**EXAMINATION COMMITTEE: Theodoros Dalamagas**, Research Director ATHENA RC
**Dimitrios Gunopulos**, Professor NKUA
**Christos Diou**, Associate Professor HUA

**Examination Date: 9th December, 2024**

**ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ**

# Αναπαράσταση Ροής Δεδομένων, Επιθεώρηση και Εξηγησιμότητα Μοντέλων σε Ροές Εργασίας Μηχανικής Μάθησης

**Στεφανία Πάτσου**
**Α.Μ.:** DS2190014

**ΕΠΙΒΛΕΠΟΝΤΕΣ: Θεόδωρος Δαλαμάγκας**, Διευθυντής Ερευνών ΕΚ ΑΘΗΝΑ

**ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: Θεόδωρος Δαλαμάγκας**, Διευθυντής Ερευνών ΕΚ ΑΘΗΝΑ
**Δημήτριος Γουνόπουλος**, Καθηγητής ΕΚΠΑ
**Χρήστος Δίου**, Αναπληρωτής Καθηγητής Χαροκόπειο Πανεπιστήμιο

**Ημερομηνία Εξέτασης: 9 Δεκεμβρίου 2024**

# ABSTRACT

MLInspect [1] is a debugging tool designed for data science pipelines. It detects issues related to data leaks, unintended data transformations and distribution shifts in ML pipelines by automatically inspecting their data flows. This tool generates Directed Acyclic Graph (DAG), which allows its users to look through all stages of pipeline processing or training. It can also observe the ML pipelines even at the time when they fail to work properly as well as debug them. Specific nodes of this DAG can have inspections attached to them so that any possible problems can be detected and diagnosed at. The latter can be either predefined or customised considering the requirements of this kind of a pipeline. Users can define custom inspections and extend the tool to meet their specific requirements. This makes MLInspect a versatile tool for a wide range of debugging and monitoring tasks.

Explainable AI (XAI) is a collection of methods and processes that allow human users to comprehend and have confidence in the results of machine learning algorithms. XAI helps describe an AI model, its impacts, and potential biases, ensuring accuracy, fairness, transparency, and reliable decision-making. There are several approaches to interpret a machine learning model. These include LIME, SHAP, Integrated Gradients, ALE, ICE etc. Each has potential of giving different outcomes about the model in question. Therefore, they should be employed considering different setup steps that could exist in an ML pipeline.

This thesis enhances MLInspect by integrating explainability features. It patches methods like LIME, SHAP and others, some described above, into a Directed Acyclic Graph (DAG) for visualisation. Additionally, it persists trained models in the DAG, facilitating subsequent inspections. When triggered, a new inspection retrieves the model image from the DAG and executes explainability methods, storing results alongside the model node. This structured approach enables seamless integration of explainability into debugging, enhancing understanding of machine learning models.

**SUBJECT AREA**: MLOps and Model Explainability

**KEYWORDS**: Explainability, Directed Acyclic Graph (DAG), Model Persistence, Inspection, Model Interpretation

# ΠΕΡΙΛΗΨΗ

Το MLInspect [1] είναι ένα εργαλείο εντοπισμού σφαλμάτων που έχει σχεδιαστεί για αγωγούς επιστήμης δεδομένων. Εντοπίζει ζητήματα που σχετίζονται με διαρροές δεδομένων, ακούσιους μετασχηματισμούς δεδομένων και αλλαγές διανομής σε αγωγούς ML επιθεωρώντας αυτόματα τις ροές δεδομένων τους. Αυτό το εργαλείο δημιουργεί το Directed Acyclic Graph (DAG), το οποίο επιτρέπει στους χρήστες του να εξετάζουν όλα τα στάδια της επεξεργασίας ή της εκπαίδευσης του αγωγού. Μπορεί, επίσης, να παρακολουθήσει τους αγωγούς επιστήμης δεδομένων ακόμη και τη στιγμή που δεν λειτουργούν σωστά, καθώς και να τα επιδιορθώνει. Συγκεκριμένοι κόμβοι του DAG μπορούν να έχουν επιθεωρήσεις που συνδέονται με αυτούς, ώστε να μπορούν να εντοπιστούν και να διαγνωστούν τυχόν προβλήματα σε σχετικά τους σημεία. Το τελευταίο μπορεί είτε να προκαθοριστεί είτε να προσαρμοστεί λαμβάνοντας υπόψη τις απαιτήσεις αυτού του είδους αγωγού. Οι χρήστες μπορούν να ορίσουν προσαρμοσμένες επιθεωρήσεις και να επεκτείνουν το εργαλείο ώστε να ανταποκρίνεται στις συγκεκριμένες απαιτήσεις τους. Αυτό καθιστά το MLInspect ένα ευέλικτο εργαλείο για ένα ευρύ φάσμα εργασιών εντοπισμού σφαλμάτων και παρακολούθησης.

Η Εξηγήσιμη Τεχνητή Νοημοσύνη (XAI) είναι μια συλλογή μεθόδων και διαδικασιών που επιτρέπουν στους ανθρώπινους χρήστες να κατανοούν και να έχουν εμπιστοσύνη στα αποτελέσματα των αλγορίθμων μηχανικής μάθησης. Η XAI βοηθά στην περιγραφή ενός μοντέλου τεχνητής νοημοσύνης, των επιπτώσεών του και των πιθανών προκαταλήψεων, διασφαλίζοντας ακρίβεια, δικαιοσύνη, διαφάνεια και αξιόπιστη λήψη αποφάσεων. Υπάρχουν διάφοροι τρόποι ερμηνείας ενός μοντέλου μηχανικής μάθησης. Αυτοί περιλαμβανομένουν το LIME, SHAP, Integrated Gradients, ALE, ICE κ.λπ. Ο καθένας έχει τη δυνατότητα να δώσει διαφορετικά αποτελέσματα σχετικά με το εν λόγω μοντέλο. Επομένως, θα πρέπει να χρησιμοποιούνται λαμβάνοντας υπόψη τα διαφορετικά στάδια εγκατάστασης που μπορεί να υπάρξουν σε έναν ML αγωγό.

Αυτή η διατριβή ενισχύει το MLInspect ενσωματώνοντας χαρακτηριστικά επεξηγηματικότητας. Εφαρμόζει μεθόδους όπως LIME, SHAP και άλλες, κάποιοι αναφέρονται και παραπάνω, σε ένα Directed Acyclic Graph (DAG) για οπτικοποίηση. Επιπλέον, διατηρεί εκπαιδευμένα μοντέλα στην DAG, διευκολύνοντας τις μετέπειτα επιθεωρήσεις. Όταν ενεργοποιείται, μια νέα επιθεώρηση ανακτά την εικόνα του μοντέλου από το DAG και εκτελεί μεθόδους επεξήγησης, αποθηκεύοντας τα αποτελέσματα παράλληλα με τον κόμβο του μοντέλου. Αυτή η δομημένη προσέγγιση επιτρέπει την απρόσκοπτη ενσωμάτωση της επεξήγησης στον εντοπισμό σφαλμάτων, βελτιώνοντας την κατανόηση των μοντέλων μηχανικής μάθησης.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ**: MLOps και Επεξήγηση Μοντέλου

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ**: Επεξηγησιμότητα, Κατευθυνόμενο Ακυκλικό Γράφημα (DAG), Αποθήκευση Μοντέλου, Επιθεώρηση, Ερμηνεία Μοντέλου

*Στον Τάκη*
*Αλλά και σε όποιον/α με στήριξε πραγματικά.*
*Σας ευχαριστώ ολόψυχα.*

# ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my thesis supervisor, Theodore Dalamagas, for his invaluable guidance, support, and patience throughout this research project. His insights and expertise were instrumental in shaping the direction and quality of this thesis.

To my companion and friend, Takis and Denia respectively, thank you for your encouragement, companionship, and the many discussions that helped me stay motivated.

Finally, I would like to extend my heartfelt gratitude to my family for their unwavering support and understanding. Your belief in me has been a constant source of strength.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LISTINGS

# 1. INTRODUCTION

Explainability in Artificial Intelligence (XAI) is a set of methods and processes that alter the way users can understand and rely on machine learning (ML) algorithms. The primary objective behind XAI is to create AI models that are transparent by revealing their operations, impacts, as well as potential biases. This kind of transparency ensures accuracy, fairness, truthfulness and dependability of artificial intelligence models, making them more reliable for decision-making. One can interpret ML models using different methods including LIME which stands for Local Interpretable Model-Agnostic Explanations, SHAP meaning Shapley Additive exPlanations, Integrated Gradients, Accumulated Local Effects (ALE), Individual Conditional Expectation (ICE) etc..

However far Explainable AI has come, there are still challenges when it comes down to implementing it into practice, especially around complex data science pipelines. Many stages are involved in processing data through pipeline systems and training models may hide bugs like leaks. These leaks can be between datasets or unintended transformations on input data or maybe distribution shifts during various stages. One great debugging tool that addresses this problem is called MLInspect which was built specifically to handle issues encountered within typical data science pipelines. It does so by creating a Directed Acyclic Graph (DAG) representation of all stages in pipeline based on automatic inspection of data flows. Users are able to follow through failed parts even if they have stopped execution since inspections can be attached at any node level with this DAG structure. This promotes successful debugging process where necessary.

To further enhance MLInspect's functionality, an extension point should be introduced, allowing advanced explanations capability integration directly into the framework itself. Such a feature would involve incorporating popular XAI methods like LIME, SHAP among others into DAG visualisation. Besides showing flow and inspections, visually enhanced version now supports saving trained models within structures representing stages, which are the nodes of the DAG, thus enabling better interaction for later execution of specific inspections against stored models. When an inspection is triggered, system retrieves stored model from DAG and runs selected explanation methods against it; then stores results of these in the corresponding DAG node.

Integrating interpretability into MLInspect ensures a better systematic way of understanding machine learning models; XAI techniques embedded within DAG allow for seamless visualisation and interpretation of behaviour or performance throughout various stages in the pipeline. These, not only help with debugging or monitoring, but also focus on understanding decisions made by ML models. These could produce a transparent view of the ML models, therefore limiting the applicability of possibly erroneous pipelines. On this note, improved vision offers comprehensive solution by acting as a powerful integrated tool for ensuring trustworthiness, reliability as well as transparency of machine learning pipelines.

This thesis is structured into 5 chapters. Chapter 2 gives some background information regarding DevOps and MLOps, their principles, known tools and a comparison between

them. It demonstrates what explainable AI is, describing the XAI methods that will be integrated in MLInspect implementation and frameworks that contain multiple XAI methods. A sample taxonomy will be also displayed, taking into account the methods used in this study. Finally, it showcases related work with the current paper. Chapter 3 reveals valuable information of MLInspect paper, which is the framework that this paper is based on. It signifies its concepts, its architecture and its prior implementation. Chapter 4 presents the current study, what was accomplished and the valuable results of the new implementation of MLInspect. Chapter 5 shows a use case demonstration of the enhanced MLInspect behaviour as well as unit tests that back up these scenarios. Finally, chapter 6 concludes the thesis, summarising what was accomplished, what could have been done differently and future improvements.

This thesis supports the advancement of tools and frameworks responsible for transparently promoting trustworthy AI systems. Explainability methods can help user pipelines become more easily portrayed, and the enhanced MLInspect can contribute more to this purpose. The ”Explainable” MLInspect framework not only improves the debugging process but also provides deeper insights into the behaviour of machine learning models.

# 2. BACKGROUND AND RELATED WORK

This section gives a comprehensive overview of theory, concepts and prior research regarding the thesis. It starts with DevOps and MLOps, it summarises its principles, it compares them, and finally reports relevant tools. The next subsection describes the Explainable AI, it categorises various explainability methods, highlights key techniques, and attempts to give a detailed example on taxonomies. In the end, it covers related work, representing two frameworks, HYPPO [2] and MLInspect. The latter promotes a better understanding of the methodologies and technologies further used in the thesis.

## 2.1 DevOps

DevOps is a software engineering concept that combines development and operation processes to ensure an improved collaboration and productivity [3]. The paper referenced gives a thorough explanation of what DevOps is and gives a more transparent and clear definition of what it encompasses. The core principles of DevOps include collaboration and communication, automation, continuous integration and continuous deployment (CI/CD), monitoring and feedback. Collaboration and communication describes the way that development and operations teams cooperate and automation outlines the reduction of manual processes by identifying the repetitive ones that could be automated. In addition, CI/CD is labeled as the method that ensures a faster release cycle of code changes. It benefits the system by getting feedback of how it operates, making it more reliable. Software Engineering Body of Knowledge (Swebok) defines the knowledge areas of DevOps. These are the CI/CD, already mentioned, automated testing which is how a code change can be tested automatically, infrastructure as code (IaC), which is managing and provisioning a system through code rather than manual efforts and lastly monitoring and logging. DevOps is a crucial modern software engineering methodology, which enables faster implementation and deployment. It can support multiple cross-functional team collaboration and transparency of code functionality.

Paper [4] illustrates a case study of DevOps in a product development organisation in New Zealand. The DevOps principles were applied and further showed that the deployment frequency was increased (from 30 to 120 releases per month) and improved the cooperation between development and operations teams.

## 2.2 MLOps

MLOps integrates machine learning with DevOps practices to create a cohesive workflow. This workflow ensures that a product/project can be more testable, scalable and more easily reproduced [5]. An example lifecycle can be viewed in Figure 2.1.

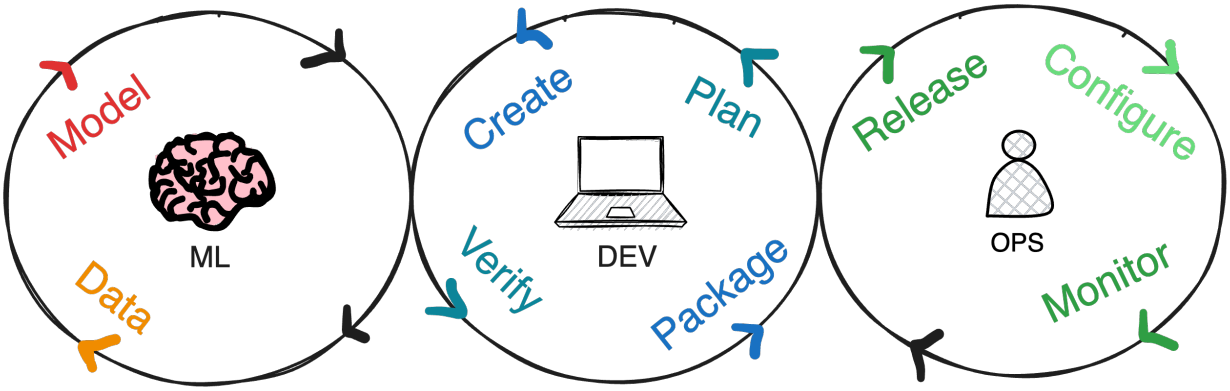



**Figure 2.1: Example MLOps Lifecycle**

### 2.2.1 Principles

MLOps is based on the principles defined in Section 2.1. They can be described as:

1. **Continuous Integration and Continuous Deployment (CI/CD)**: This principle is all about automating the entire ML pipeline, from gathering data to deploying models. Continuous integration is applied to enable changes in the codebase to be regularly merged and tested. Continuous deployment also contributes to ensuring that the latest model versions are automatically deployed to production environments. This automation allows for a reduced amount of manual intervention cases and faster deployment that offers better reliability.

2. **Versioning**: Versioning is actually used to flag a deployment with a specific identifier. This additional information makes it easy for a developer or a third party community member to trace the artefacts deployed, including data, models, and changed code. This, furthermore, enables them to track changes, replicate experiments, and understand the evolution of models over time. Model development and deployment is more easily tracked down, offering a sort of audit and lineage mechanism of the model.

3. **Testing**: MLOps offers a better way to test the model defined. Using rigorous testing protocols during CI workflow, a developer can validate further a model and assess its performance. Testing includes unit tests, integration tests for pipeline components, and validation tests for model accuracy and robustness. With a testing suite during CI, there is now a way to identify possible errors of the model before a model is deployed to production.

4. **Monitoring**: Comprehensive monitoring is implemented in order to track model performance and data drift in production environments. There are monitoring tools that evaluate the model's predictions and performance metrics and if something is out of the ordinary, these tools can alert subscribed teams for any deviations or anomalies. This monitoring capability is utilised when a model has been deployed to production

or at a prior phase, at a UAT-session, which is a pre-production environment that simulates production. This proactive approach allows for timely interventions when necessary.

5. **Collaboration**: MLOps requires that there is collaboration among various operational members of a team, such as data scientists, engineers and operations. Different domains are combined into the ML lifecycle by encouraging cross-functional collaboration. They help each other, ensuring that the models defined are characterised by quality, accuracy and improved performance.

6. **Governance and Compliance**: This is where governance frameworks are put in place to ensure adherence of models to legal and ethical guidelines. Data privacy and particularly GDPR are significant components that must be integrated within an MLOps life cycle's workflow for trust creation purposes. Transparency in operations will also enhance decision making while ensuring accountability throughout an ML lifecycle. Successful governance promotes responsible deployment of models.

Additionally, an MLOps workflow is consisted of four phases:

1. **Data Engineering**: The first stage of an MLOps workflow is data engineering. This entails preparing and preprocessing data before it is fed into the model during training. Data scientists collect all data they require from available sources. These include both internal and external ones which could be either structured or unstructured types of data formats depending on their research questions' requirements. Such datasets go through cleaning and transforming before being stored, thus the whole process has to involve sound data engineering practices for improved accuracy and reproducibility.

2. **Model Training**: Model training, as the name implies, involves implementing and training machine learning models. Choosing the best model algorithms or techniques during this phase consists of tuning hyper-parameters, and model performance validation. This section can be revisited several times because it is very difficult to find the optimal model after one iteration of the stages. These models are appropriately tested with unit tests or integration tests to allow finding the ideal algorithm and parameters.

3. **Model Deployment**: Model deployment to production environment can be accomplished through CI/CD pipelines without hitches, accommodating a smooth transition into the latest version of the model. By doing this, automation cuts down on deployment time, reduces errors and ensures that models are always updated in a consistent and reliable manner. Also, as previously mentioned, in this stage, an identifier depicting the version of the model is stored, to keep track of the model that was deployed.

4. **Model Monitoring**: A well built MLOps workflow includes continuously monitoring deployed models. Key indicators like performance, time to consume the whole training set and if a past version was more accurate than the latest one are tracked by

monitoring tools. This stage also includes alerting of metrics that cross a defined threshold, that probably introduce anomalies. Teams getting these alerts can more easily gather all possible issues and continue with their resolution. The model is always assessed which enables faster adjustments with visualisation of errors and key indicators.

A more detailed workflow can be represented in Figure 2.2. The workflow depicted shows how each component of MLOps works and how a phase interacts with others.

### 2.2.2 DevOps vs MLOps

DevOps is a practice aimed at improving development cycles, deployment speed, and release reliability in large-scale software systems through Continuous Integration (CI) and Continuous Delivery (CD) [6].

However, ML systems have many differences from the software systems. First of all, ML projects involve data scientists focused on exploratory data analysis and model development, who may lack experience in building production-grade services. In addition, ML development is experimental, requiring extensive testing of features, algorithms, and models, with a focus on reproducibility and code reuse. They also need to be tested multiple times, including data validation, model quality evaluation, and model validation, beyond typical unit and integration tests. Deploying ML systems involves complex pipelines for automatic retraining and deployment and unlike straightforward software deployment in production environment, ML models can degrade over time due to evolving data, necessitating continuous monitoring and potential rollbacks. ML systems specifically adjust their CI/CD processes. For example, CI includes testing and validating data, schemas, and models, whereas, CD involves deploying an ML training pipeline that deploys a model prediction service. Continuous Training (CT), which is not part of software systems, focuses on the automated retraining and serving of models.

Overall, DevOps principles apply to ML systems, but with significant adjustments that handle the unique challenges of ML development, deployment, and production monitoring. ML operations require a structured approach to experimentation, tracking, and maintaining model performance over time.

### 2.2.3 Tools

All these years, many tools have been implemented to accommodate the MLOps principles. A segment of some tools is considered in Table 2.1.

**Table 2.1: Example MLOps Tools**

| Type | Name | Description |
|---|---|---|
| Automation and Orchestration | Airflow | Designed for creating, scheduling, and overseeing batch-oriented workflows. Allows the development of workflows that integrate with nearly any technology. |
| Automation and Orchestration | Luigi | It helps build complex pipelines of batch jobs. It handles dependency resolution, workflow management, visualisation, handling failures, command line integration and more. |
| CI/CD | CML | It is an open-source CLI tool for implementing continuous integration & delivery (CI/CD) with a focus on MLOps. It automates development workflows, including machine provisioning, model training and evaluation, compares ML experiments across project history, and monitors changing datasets. |
| CI/CD | Aqueduct | It is an MLOps framework that allows definition and deployment of machine learning and LLM workloads on any cloud infrastructure. |

| | | |
|---|---|---|
| Data Management | DVC[7] | It is a command line tool that helps develop reproducible machine learning projects. It versions data and models, it stores them in the cloud storage but keep their version info in Git repo and it compares any data, code, parameters, model, or performance plots. It also shares experiments. |
| Data Management | Delta Lake | An open-source storage framework that enables building a Lakehouse architecture [8] with compute engines including Spark, PrestoDB, Flink, Trino, and Hive and APIs for Scala, Java, Rust, Ruby, and Python. |
| Data Management | Feast | It is an open-source feature store for machine learning. It manages existing infrastructure to productiosize analytic data for model training and online inference. |
| Data Management | Pachyderm | Automates complex pipelines with sophisticated data transformations. It versions large data sets of unstructured and structured data automatically, intelligently with a git-like structure. |
| Experiment Tracking | Comet | Integrates with existing infrastructure so as to manage, visualise, and optimise models. It handles models from training up to production. |

| | | |
|---|---|---|
| Experiment Tracking | MLflow | It is an open-source platform that focuses on the full lifecycle for machine learning projects, ensuring that each phase is manageable, traceable, and reproducible. |
| Experiment Tracking | Neptune[9] | It offers a single place to track, compare, store, and collaborate on experiments and models. |
| Experiment Tracking | Weights & Biases | It trains, fine-tunes and manages models from experimentation to production, and track and evaluate LLM applications. |
| Model Deployment | BentoML[10] | It is an open-source model serving framework that simplifies how AI/ML models gets into production. |
| Model Deployment | Kubeflow | It makes deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable. |
| Model Deployment | TensorFlow Serving | It makes it easy to deploy new algorithms and experiments, while keeping the same server architecture and APIs. TensorFlow Serving provides out-of-the-box integration with TensorFlow models, but can be easily extended to serve other types of models and data. |
| Model Deployment | TorchServe | It is a performant, flexible and easy to use tool for serving PyTorch models in production. |

| | | |
|---|---|---|
| Monitoring | Evidently | Evidently helps evaluate, test, and monitor data and ML-powered systems. It offers predictive tasks, generative tasks and data monitoring. |
| Visualization | Plotly | Plotly is a python graphing library that makes interactive, publication-quality graphs. |
| Visualization | Altair[11, 12] | Vega-Altair is a declarative statistical visualisation library for Python. Vega-Altair helps understand the data and their meaning. |

Awesome Production Machine Learning repository [13] offers a list of multiple tools that can be explored further.

## 2.3 Explainable AI

Explainability in AI is the ability to illustrate, in an understanding manner, the way machine learning models are behaving, what decisions they are making and their output. This is important as it helps build trust and ensures fairness and accountability in AI systems. Most importantly, explainability assists users to comprehend and trust AI by providing transparency on how decisions were made. Secondly, Explainability also plays a part in uncovering and addressing biases, promoting ethical use of technology and ensuring compliance with regulations [14].

However, there are some difficulties associated with explainability. A great deal of AI models, especially deep learning models, are like “black boxes” that are not easily interpretable from within. Crossing the tight rope between complexity of model versus interpretability forms a big challenge; simpler models are easy to understand but might lack accuracy as compared to more complex ones. For example, healthcare can use explainability to diagnose diseases through transparently stating the reasons behind its decisions. In finance, it may be used for credit scoring by explaining why a loan was approved or denied. Explainability can also be applied in the legal field, autonomous driving and other high-stakes areas where understanding AI decisions is critical.

Research is currently concentrated on developing more sophisticated techniques that balance interpretability and accuracy. User-centric explainability, which focuses on how explanations are given to end users to ensure they are understandable and actionable, is becoming increasingly important.

### 2.3.1 Taxonomy

Categorising explainability methods will facilitate the implementation of a more concrete understandability of explainability methods. For this purpose, articles [15] and [16] will be considered as well as the paper with title ”A Review of Taxonomies of Explainable Artificial Intelligence” [17]. In the paper, it is explained how the explainability methods can be classified regarding 4 approaches that are followed in the ML industry. The first classification is the Functioning-Based Approach which focuses on how the methods work, e.g. local perturbations, leveraging structure, meta-explanations, architecture modification, and extracting examples. The second approach, Result-Based, concentrates on the result that the methods may have, either feature importance or surrogate models and more. The third approach is the Conceptual. This approach is based on conceptual objectives such as transparency, interpretability and trustworthiness. Finally, the last approach is the Mixed one, that brings together aspects from other approaches to provide a more holistic taxonomy. The suggested taxonomy can be outlined in Figure 2.3.

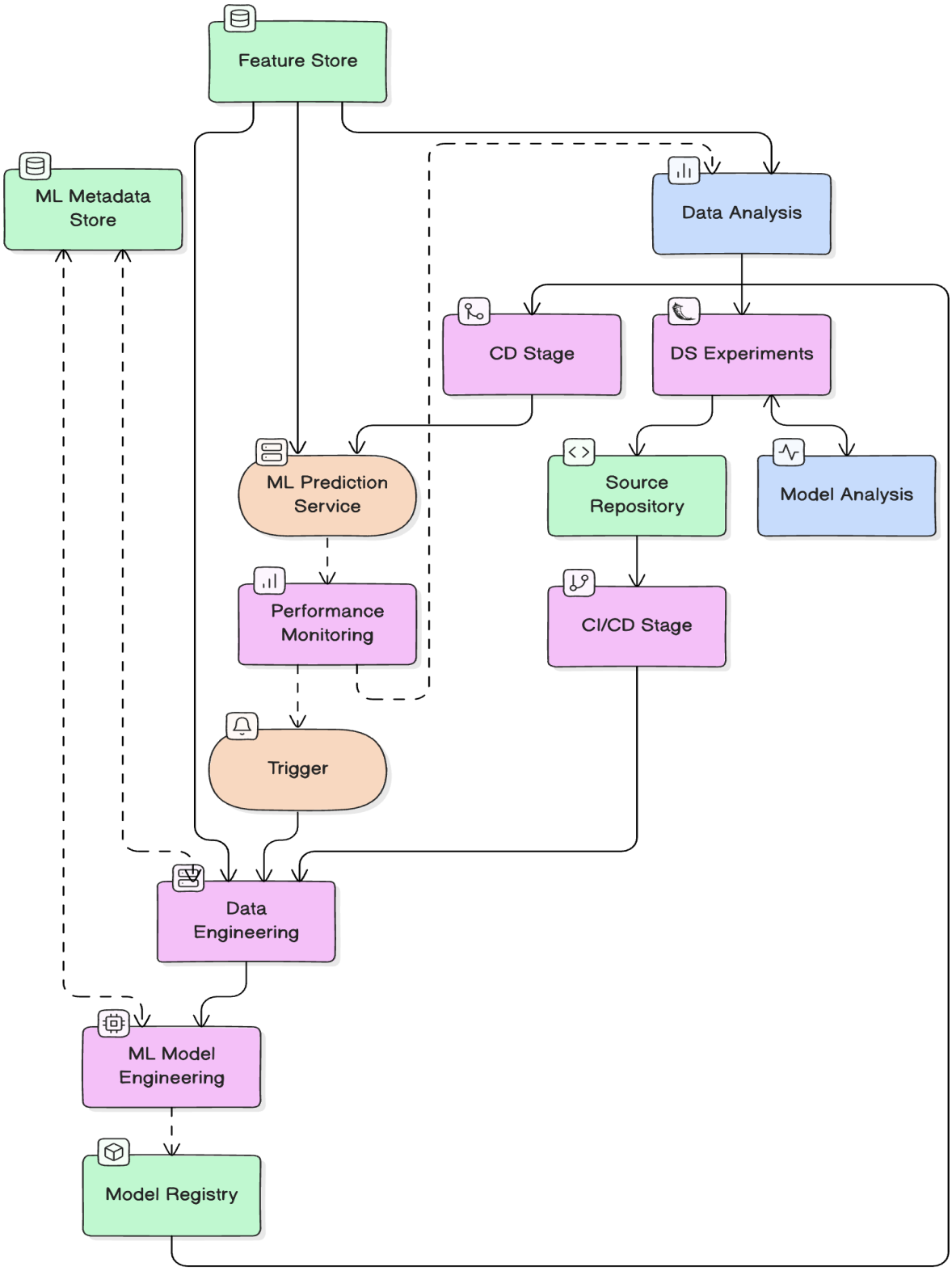



**Figure 2.2: Example MLOps Detailed Workflow**

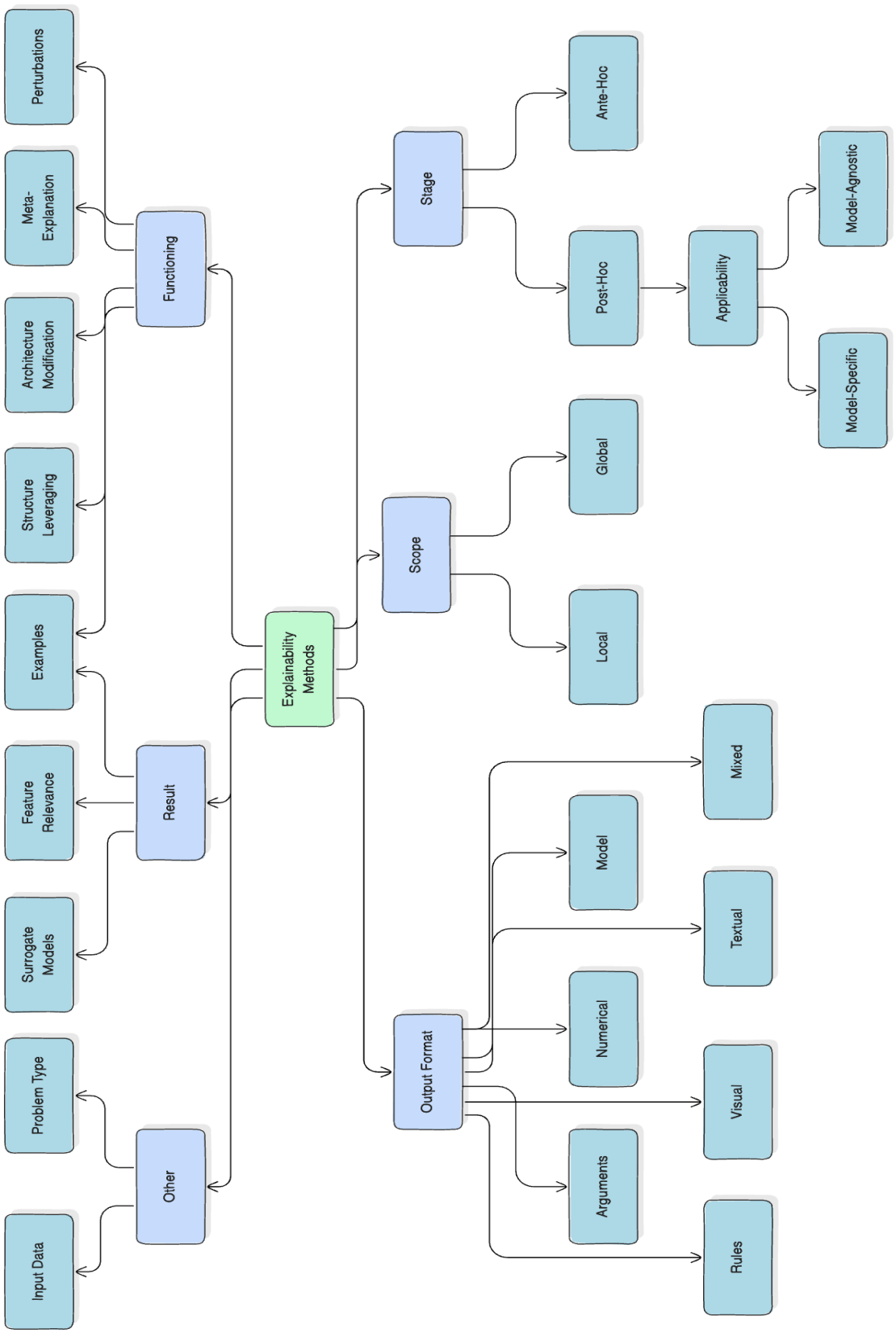



**Figure 2.3: Proposed Taxonomy of XAI Methods**

As shown in Figure 2.3, XAI methods can be divided in multiple categories. There can be classified regarding their result, their output format, their scope, their stage, their functionality and other factors.

**Result.** XAI methods can be categorised as to the end result. It highlights different approaches for deriving explanations based on the end result, like feature relevance or leveraging model structure. Feature Relevance methods assess the importance of features in the model. Examples methods use examples to explain model behaviour. Lastly, Surrogate Models methods use simpler models to approximate and explain complex models.

**Functioning.** Meta-Explanation methods explain further other explanations. In addition, Structure Leveraging methods leverage the structure of the model for explanations. Architecture Modification methods involve modifying the model architecture to improve explainability. Examples methods are defined above and depict the same functionality. Finally, Perturbations methods use perturbations in the input data to generate explanations.

**Other.** This category includes Input Data and Problem Type methods. Input Data methods explain by examining the input data. Problem Type methods vary based on the type of problem being solved (e.g. classification, regression). The model output can alter how the explainability method works. A model, for example, can either output probabilities of a prediction for each data point or classify the data points regarding these probabilities.

**Output Format.** This classification includes the output format of an XAI method. It can output arguments, rules, numerical values, or visual explanations in the form of plots and graphs. It can be a new model (e.g. surrogate model), textual explanations, or mixed, offering multiple formats.

**Scope.** This class of taxonomy is comprised of local and global types. Local ones provide explanations for individual predictions. Local explainability methods refer to the ones that will describe how a single datapoint led to a specific result. Global scope includes methods that provide explanations for the entire model. It will give us a bigger picture of how each feature collectively contributes to the results (predictions) of a model. In other words, it will give us an explanation in general regarding the model using many data points.

**Stage.** Stage is split into Post-Hoc and Ante-Hoc categories. Post-Hoc methods provide explanations after the model has been trained. Ante-Hoc methods are built into the model from the beginning to provide inherent explainability. Post-Hoc is further divided into Model-Specific and Model-Agnostic classes. Model-Specific refers to XAI methods that are tailored for specific types of models or group of models. On the other hand, Model-Agnostic are fit for any model type. It is simply methods that do not take into account the model trained (whether it is a linear, NN, tree-based etc.).

This taxonomy can be applied to categorise efficiently various XAI methods, helping users identify easier the diverse complex methods that exist.

### 2.3.2 Popular Methods

There is a considerable amount of XAI methods that have been contributed from other data scientists in the community. More are provided as a result of a comprehensive scientific research that was completed and reviewed. Coefficients of Logistic Regression [18], Anchors [19], Contrastive Explanations Method (CEM) [20], Counterfactual Instances [21, 22, 23], Deep Learning Important Features (DeepLIFT) [24], Explainable Boosting Machine (EBM) [25], Global Interpretation via Recursive Partitioning (GIRP) [26], Layer-wise relevance propagation (LRP) [27], Leave One Covariate Out (LOCO) [28], Morris Sensitivity Analysis [29], ProfWeight [30], ProtoDash [31], Scalable Bayesian Rule Lists [32] and Tree Surrogates [33, 34] are some of the XAI methods that can be further inspected and studied.

In the next sections, the most popular XAI methods will be defined, which will also be the ones that are going to be integrated in the Explainable MLInspect project.

#### 2.3.2.1 LIME

Local interpretable model-agnostic explanations (LIME) [35] is an approach that fits a surrogate glass-box model around any black-box model's prediction space. LIME aims at modelling explicitly every local neighbourhood around each prediction made by such black-box systems. It works through perturbing individual data points before generating synthetic data which get evaluated within those black-box systems, before being finally utilised as training set for corresponding glass-box models. This helps users understand which features affect particular predictions and make their outputs more transparent and reliable.

The main focus of LIME is to interpret individual predictions by creating a local surrogate model that is interpretable, making it easier to explain the specific instance for which an explanation is required. LIME works by initially selecting an instance to explain the prediction of the model. This can be a specific data point of a dataset. Then, a dataset is created with perturbed instances by making minor changes to the original instance. These perturbations are usually small modifications of the initial state of the selected instance, regarding the feature values. These perturbations are then used as a sample dataset to the original complex model to get the predictions for each one. This step is very important, as weight is assigned to each perturbed instance, based on its similarity to the original instance. Typically, a kernel function is used for this goal. It usually gives higher weights to an instance that is more similar to the original one. After that, a new interpretable model is created, e.g. linear or decision tree, and it is trained on the perturbed dataset, using the weights defined before, focusing more to the perturbations that are closer to the original instance. Finally, the coefficients or structure of the interpretable model give insights as to which feature was the most important, regarding the impact on the prediction for the original instance.

Let's illustrate LIME with a simple example. In this example, we have a black-box model

predicting whether a person has diabetes based on two features: age and glucose level. We want to explain the prediction for a specific patient. The patient data are: $Age = 50, GlucoseLevel = 150$ The model data that we own is that it predicts a 70% probability of diabetes for this patient. The first step is to select an instance to explain. This is $x = (50, 150)$. For this specific instance, perturbations are generated by slightly tweaking age and glucose level. These can be:

- $x\_1 = (49, 149)$
- $x\_2 = (51, 151$
- $x_3 = (48, 150)$
- $x_4 = (52, 148)$

Then, using the black-box model, the predictions we obtain for the perturbations are:

- Prediction for $x_1$: 65% probability of diabetes.
- Prediction for $x_2$: 72% probability of diabetes.
- Prediction for $x_3$: 68% probability of diabetes.
- Prediction for $x_4$: 71% probability of diabetes.

Using these predictions, the distances and weights are calculated for these perturbed data.

- Distance between $x$ and $x_1$: 1.41, Weight: 0.95.
- Distance between $x$ and $x_2$: 1.41, Weight: 0.95.
- Distance between $x$ and $x_3$: 2.00, Weight: 0.85.
- Distance between $x$ and $x_4$: 2.83, Weight: 0.75.

The weights calculated in the previous step are then utilized in a linear model definition. The linear model will be specified as: $DiabetesProbability = 0.3 \times Age + 0.5 \times GlucoseLevel$. Finally, the new linear model is interpreted using the predictions of the perturbed data. For example, the coefficients of the linear model suggested that a 1-year increase in age increases the diabetes probability by 0.3 and a 1-unit increase in glucose level increases the diabetes probability by 0.5.

#### 2.3.2.2 Shapley Values

SHAP (SHapley Additive exPlanations) [36] is a framework that explains the output of any model using Shapley values, a game-theoretic approach often used for optimal credit allocation. While this can be used on any black-box model, SHAP can compute more efficiently on specific model classes (like tree ensembles). It allows optimal credit allocation and local explanations which simplify the interpretation of complex models. It was named after Lloyd Shapley and it ensures that each feature's contribution to the model's prediction is adequately quantified.

Each feature of a dataset is treated as a "player" in a game where the goal is to achieve the prediction of the model. The Shapley value for a feature symbolizes the average contribution of that feature across all possible subsets of features. This method considers the contribution of each feature in the context of every possible combination of other features. The first step of computing Shapley values is to look at all possible subsets (coalitions) of the feature set. Then, for each subset, calculate the marginal contribution of the feature of interest by including and excluding it from the subset. Finally, the Shapley value is the average of these marginal contributions across all possible subsets.

Mathematically, for a feature $i$, the Shapley value $\phi_i$ is calculated as:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} (v(S \cup \{i\}) - v(S))$$

Where: $N$ is the set of all features, $S$ is a subset of $N$ that does not include $i$ and $v(S)$ is the prediction for subset $S$.

A simple example will give a better insight as to how Shapley values work. The example will consider a model that predicts the house price based on three features: location, size, and age. What we want is to explain the prediction for a specific house. This house's data is: $Location = "Urban", Size = 150m^2, Age = 10years$ with a model prediction of: 300,000€.

The steps are:

1. **Coalition Formation**: Create all possible subsets of the three features: {}, {Location}, {Size}, {Age}, {Location, Size}, {Location, Age}, {Size, Age}, and {Location, Size, Age}.
2. **Marginal Contribution Calculation**: For each subset, include and exclude each feature and calculate the prediction difference.
3. **Example Subset Calculations**:
   - Subset = {}: Prediction = 250,000€.
   - Subset = {Location}: Prediction = 270,000€.
   - Subset = {Size}: Prediction = 260,000€.

- Subset = {Age}: Prediction = 255,000€.
- Subset = {Location, Size}: Prediction = 290,000€.
- Subset = {Location, Age}: Prediction = 275,000€.
- Subset = {Size, Age}: Prediction = 265,000€.
- Full set = {Location, Size, Age}: Prediction = 300,000€.

For feature ”Location”:

- Contribution when added to {}: 270,000 - 250,000 = 20,000€.
- Contribution when added to {Size}: 290,000 - 260,000 = 30,000€.
- Contribution when added to {Age}: 275,000 - 255,000 = 20,000€.

The average contribution for Location: $\frac{20{,}000+30{,}000+20{,}000}{3} = 23,333.33$€.
For feature ”Size”:

- Contribution when added to {}: 260,000 - 250,000 = 10,000€.
- Contribution when added to {Location}: 290,000 - 270,000 = 20,000€.
- Contribution when added to {Age}: 265,000 - 255,000 = 10,000€.

The average contribution for Size: $\frac{10{,}000+20{,}000+10{,}000}{3} = 13,333.33$€.
For feature ”Age”:

- Contribution when added to {}: 255,000 - 250,000 = 5,000€.
- Contribution when added to {Location}: 275,000 - 270,000 = 5,000€.
- Contribution when added to {Size}: 265,000 - 260,000 = 5,000€.

The average contribution for Age: $\frac{5{,}000+5{,}000+5{,}000}{3} = 5,000$€.
These contributions sum to the total prediction:

$$23,333.33 + 13,333.33 + 5,000 = 41,666.67€$$

which should be normalised to match the model prediction of 300,000€.

#### 2.3.2.3 Integrated Gradients

Integrated Gradients [37] method seeks to attribute importance value for each input feature of a machine learning model based on gradients of the model output with respect to the input. It was introduced by Sundararajan, Taly, and Yan in 2017 to address the challenge of understanding the inner workings of neural networks. It has many use cases including understanding feature importance, identifying data skew and debugging model performance. The technique is useful for regression and classification models. For non-scalar outputs, like multi-target regression or classification models, gradients are calculated on one particular element in the output. In case of a classification model, usually the output's gradient refers to either the true class or class predicted by the model.

The calculation of Integrated Gradients can be done by firstly setting up a baseline input, which is usually an input where all features are set to zero or some other neutral value. This baseline works as a point of reference from which the change in the model's output can be measured. Then, the gradients of the model's output with respect to the input features are calculated. These gradients are integrated along a straight-line path from the baseline input to the actual input. Finally, the attribution score for each feature is computed. This score is the integral of the gradients over the path mentioned and depicts the contribution of each feature to the model's prediction.

In a mathematical form, given a model $F$ and an input $x$, the integrated gradient along the $i$-th dimension is defined as:

$$IG_i(x) = (x_i - x_i') \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha(x - x'))}{\partial x_i} d\alpha$$

where $x'$ is the baseline input, $x_i$ is the $i$-th feature of the actual input, and $\alpha$ is a scalar that scales the input from the baseline to the actual input.

An example that encapsulates the Integrated Gradients implementation can be a simple neural network model that predicts the price of a house based on three features: size (in square meters), number of bedrooms, and age (in years). Integrated Gradients will be used to understand how each feature contributes to the predicted price.

Steps of calculation:

1. **Baseline Input**: We choose a baseline input where the size, number of bedrooms, and age are all set to zero:
$$baseline = [0, 0, 0]$$
2. **Actual Input**: Assume that our actual input for a specific house is:
$$actualinput = [200, 3, 20]$$
3. **Model Prediction**: Our neural network model $F$ predicts the price of the house based on these features.

4. **Path Integration**: In this step, the gradients of the model's output with respect to each feature are calculated, and integrated along the path from the baseline to the actual input.

5. **Attribution Scores**: The integrated gradients give us the attribution scores for each feature. Let's say we compute the following scores:

$$IG_{size} = 100, \quad IG_{bedrooms} = 50, \quad IG_{age} = -10$$

These scores indicate that the size of the house contributes 100 units to the predicted price, the number of bedrooms contributes 50 units, and the age of the house subtracts 10 units from the predicted price.

#### 2.3.2.4 Accumulated Local Effects (ALE)

Accumulated Local Effects (ALE) [38] are measures that show the effects of features on machine learning model predictions. In the alternative to partial dependence plots, which are slow and biased, ALE plots serve as faster and fairer solutions. The PDPs obtain an average effect of a feature over its entire range while ALE plots examine how local changes in the prediction as a feature value varies.

The ALE method starts by calculating the local effect of a feature, by measuring the change in the prediction when the feature changes. All other features remain unchanged. This calculation is averaged over all instances. After that, the local effects are accumulated to show how the prediction changes as the feature varies across its range. This aggregation helps in visualising the feature's impact on the model's predictions, in a more global manner. Finally, the ALE plot is centred to have an average effect of zero. In this way, if any deviations from the average prediction exist, they will be shown in the plot.

In a mathematical perspective, given a feature $X_j$ and a model $f$, the ALE for feature $j$ at a value $x_j$ is defined as:

1. Divide the range of $X_j$ into intervals. These intervals can be quantiles or evenly spaced.

2. Compute the local effect. For each interval, the local effect is computed by taking the difference in the model's prediction at the interval boundaries, averaged over all instances that fall within the interval.

3. Accumulate the local effects. The local effects are summed up to give the accumulated effect, which is then centered. The accumulated effect for feature $j$ at a point $x_j$ is given by:

$$ALE_j(x_j) = \sum_{k=1}^{m} \left( \frac{1}{|N_k|} \sum_{i \in N_k} \left[ f(x_{boundary}^{(k+1)}) - f(x_{boundary}^{(k)}) \right] \right)$$

where $N_k$ is the set of instances in the $k$-th interval, and $x_{boundary}^{(k)}$ are the boundaries of the $k$-th interval.

Let's consider, again, a simple example of predicting house prices. The model uses three features: size (in square meters), number of bedrooms, and age (in years).

The steps are:

1. **Model and Data**: The hypothesis is that we have trained a regression model to predict house prices. Our dataset includes the mentioned features.
2. **Divide the Range of the Feature**: We will focus on the "size" feature. We divide the range of "size" into several intervals. For simplicity, we will use three intervals: 0-100, 100-200, and 200-300 square meters.
3. **Compute Local Effects**: For the first interval (0-100 sqm), we calculate the average prediction difference when the size increases from 0 to 100 square meters. For the second interval (100-200 sqm), we calculate the average prediction difference when the size increases from 100 to 200 sqm. For the third interval (200-300 sqm), we calculate the average prediction difference when the size increases from 200 to 300 sqm. Supposedly, their predictions are 50,000, 30,000 and 20,000 respectively.
4. **Accumulate and Center the Effects**: We aggregate these local effects and then centre the ALE plot to have an average effect of zero. To create the ALE plot, we accumulate these effects: ALE for 0-100 sqm: 50,000, ALE for 100-200 sqm: 50,000 + 30,000 = 80,000 and ALE for 200-300 sqm: 80,000 + 20,000 = 100,000. To centre the plot, we subtract the mean of these accumulated effects. The mean is $(50,000 + 80,000 + 100,000)/3 = 76,667$. The centred ALE values are:
   - $0 - 100sqm : 50,000 - 76,667 = -26,667$
   - $100 - 200sqm : 80,000 - 76,667 = 3,333$
   - $200 - 300sqm : 100,000 - 76,667 = 23,333$

#### 2.3.2.5 Partial Dependence Plots (PDP)

A PDP illustrates the connection between target variable of interest and any or several features in a machine learning model. The users can understand the impact on predicted outcomes because specific feature values change while averaging out the impacts of other features. In particular, this is useful for interpreting complex non-linear models [39].

PDPs show in a global manner the effect of one or two features on the predicted outcome, averaged across all the instances in the dataset. They also depict the marginal effect of a feature and in the same time the influence of all other features is averaged out. This helps in isolating the effect of the feature of interest. Lastly, PDPs are model-agnostic, meaning that they can be applied to any machine learning model, regardless of its complexity or structure.

Given a model $f$ and a feature $X_j$, the partial dependence function for $X_j$ is defined as:

$$\hat{f}_j(x_j) = \frac{1}{n} \sum_{i=1}^{n} f(x_j, X_{-j}^{(i)})$$

where $n$ is the number of instances, $X_{-j}^{(i)}$ represents all other features except $X_j$ for instance $i$, and $x_j$ is the value of the feature $X_j$ being analysed.

PDP plots can be displayed by following the below steps:

1. We choose one or two features that we want to visualise the partial dependence for.
2. We define a grid of points over the range of the selected feature(s).
3. We compute the model's predictions while varying the selected feature(s) and averaging out the other features. This is done for each point in the grid.

The average predicted outcome as a function of the selected feature(s) is being plotted, which is also the desired PDP.

An easy example is the following. We implemented a machine learning model that predicts house prices. We will consider three features: size (in square feet), number of bedrooms, and age (in years). We will focus on the ”size” feature.

1. In our hypothesis, we have trained a regression model to predict house prices. We have a dataset with instances like:
   - Instance 1: size = 120 sqm, bedrooms = 3, age = 10 years.
   - Instance 2: size = 180 sqm, bedrooms = 4, age = 5 years.
   - Instance 3: size = 240 sqm, bedrooms = 4, age = 15 years.
2. We choose the ”size” feature to visualize its partial dependence on the predicted house price. For this reason, we define a grid of points over the range of ”size” from 100 to 300 sqm in increments of 50 sqm. We vary the ”size” feature across the grid points: 100, 150, 200, 250, and 300 sqm.
3. For each point in the grid, we compute the model's predictions while varying the ”size” feature and averaging out the other features. Specifically, for the above instances and with assumed predictions:
   - At size = 100 sqm: average prediction = $\frac{1}{3}(200,000 + 250,000 + 230,000) = 226,667$.
   - At size = 150 sqm: average prediction = $\frac{1}{3}(220,000 + 270,000 + 250,000) = 246,667$.
   - At size = 200 sqm: average prediction = $\frac{1}{3}(240,000 + 290,000 + 270,000) = 266,667$.

- At size = 250 sqm: average prediction = $\frac{1}{3}(260,000 + 310,000 + 290,000) = 286,667$.
- At size = 300 sqm: average prediction = $\frac{1}{3}(280,000 + 330,000 + 310,000) = 306,667$.

The PDP plot has as X-axis the ”size” feature and the average predicted house price on the Y-axis.

Firstly, The PDP plot shows the average effect of ”size” on the predicted house price. By increasing the size from 100 to 300 sqm the average predicted price increases as well, from 226,667 to 306,667. The plot is monotonic, the predicted house price changes with the same rate as the size of the house. In the above example, a PDP plot reveals that larger houses generally have higher predicted prices, which aligns with domain knowledge. Finally, with a PDP plot, we can interpret the model behaviour, giving insights as to how the model operates and if the predictions are expected.

#### 2.3.2.6 Individual Condition Expectations plots (ICE)

An ICE plot displays the predicted outcome against the values of a feature for each instance in the dataset [40]. Unlike PDPs, which show the average effect of a feature, ICE plots show the effect for each individual instance, providing insights into the heterogeneity of the feature's influence.

ICE plots show the change in prediction for each instance as a feature varies. In this way, a more personalised view can be created, compared to the aggregated view in PDPs. Taking into account the values of the features being analysed, this method focuses on the conditional expectation of the model's prediction regarding these values. Then, the individual effects for multiple instances are plotted, revealing any variations in the feature influence across different instances.

Mathematically, given a model $f$, a feature $X_j$, and an instance $i$ with feature values $X_{-j}^{(i)}$ (all features except $j$), the ICE curve for instance $i$ is defined as:

$$ICE_i(x_j) = f(x_j, X_{-j}^{(i)})$$

where $x_j$ varies across its range, and $X_{-j}^{(i)}$ remains fixed for instance $i$.

The steps to eventually visualise the ICE plots are:

1. Choose a sample of instances from the dataset.
2. For each instance, vary the feature $X_j$ across its range while keeping other features fixed.
3. Compute the model's predictions for each value of $X_j$ for each instance.

4. Plot the individual prediction curves for each instance, showing the effect of $X_j$ on the predictions.

Let's say we have a machine learning model that predicts house prices. There are three features to consider: size (in square meters), number of bedrooms, and age (in years). We will focus on the "size" feature.

1. **Model and Data**: We have trained a regression model to predict house prices.
2. **Select Instances**: We select three instances from our dataset:
   - Instance 1: size = 120 sqm, bedrooms = 3, age = 10 years
   - Instance 2: size = 180 sqm, bedrooms = 4, age = 5 years
   - Instance 3: size = 240 sqm, bedrooms = 4, age = 15 years
3. **Vary the Feature**: We vary the "size" feature from 100 to 300 sqm in increments of 50 sqm while keeping the other features fixed.
4. **Compute Predictions**: We compute the model's predictions for each value of "size" for the selected instances.
5. **Plot the Curves**: We plot the individual prediction curves for each instance, showing the effect of "size" on the predictions.

   For Instance 1 (size = 100 sqm, bedrooms = 3, age = 10 years): Predictions for size = 100, 150, 200, 250, 300 sqm: [200,000, 220,000, 240,000, 260,000, 280,000].

   For Instance 2 (size = 180 sqm, bedrooms = 4, age = 5 years): Predictions for size = 100, 150, 200, 250, 300 sqm: [250,000, 270,000, 290,000, 310,000, 330,000].

   For Instance 3 (size = 240 sqm, bedrooms = 4, age = 15 years): Predictions for size = 100, 150, 200, 250, 300 sqm: [230,000, 250,000, 270,000, 290,000, 310,000].

Then, the ICE plot can be visualized as a line plot where the X-axis will be the "size" feature, range from 100 to 300 sqm and the Y-axis are the model's predicted prices.

#### 2.3.2.7 DALE

The Accumulated Local Effect (ALE) is a more accurate feature impact estimation technique that resolves an underlying issue in earlier methods such as Partial Dependence Plots. Nonetheless, the well-known approximation of estimating ALE from a small training set has two main limitations. First, it is not suitable for high-dimensional input data. Furthermore, instances of out-of-distribution (OOD) sampling may be encountered when there are few samples in the training set. There are cases where DALE [41] can be applied if differentiable machine learning model exists and an auto-differentiation framework

is available. DALE provides an unbiased estimator of ALE and also a procedure for quantifying the standard error of explanation.

Feature effect methods include the already mentioned Partial Dependence Plots (PDP), Marginal Plots (MPlots), but also ALE and they quantify what individual features do to the model's output. However, PDPs and MPlots fail with correlated features hence leading to inaccurate estimations. Using auto-differentiation enables DALE to overcome some limitations of ALE by allowing efficient computation of feature effects directly from gradients of the model. This ensures only observed data points are used, avoiding generation of artificial samples that can result in OOD sampling issues. The approximation formula for ALE by DALE is given below:

$$f_{DALE}(x_s) = \Delta x \sum_{k=1}^{k_x} \frac{1}{|S_k|} \sum_{i:x_i \in S_k} f_s(x_i)$$

Where $f_s(x_i)$ is the partial derivative with respect to the feature $x_s$. DALE computes local effects very efficiently using a single pass, which reduces the high computational overhead compared to traditional ALE. It scales well with big training sets and high-dimensional datasets, thus making it appropriate for complex machine learning models. Additionally, DALE allows users to change feature effect plots at different resolutions without having to redo all approximations.

To illustrate how effective DALE is, a simple example should be given. Suppose we have a dataset with two correlated features, $x_1$ and $x_2$, and a black-box model $f$. The mathematical expression of the model's output is defined as:

$$f(x_1, x_2) = 1 - x_1 - x_2$$

if $x_1 + x_2 \leq 1$ else $f(x_1, x_2) = 0$.

The features are uniformly distributed with $x_1 \sim U(0, 1)$ and $x_2 = x_1$.

1. **Partial Dependence Plot (PDP)**: PDP would mistakenly calculate the effect of $x_1$ due to its integration over unrealistic instances, resulting to a quadratic effect.
2. **Marginal Plot (MPlot)**: MPlot also overestimates the combined effect of $x_1$ and $x_2$ in the linear subregion, thus providing misleading information.
3. **Accumulated Local Effects (ALE)**: ALE would provide a more accurate effect by considering the conditional distribution $X_c|X_s = z$.
4. **DALE**: By leveraging auto-differentiation, DALE efficiently computes the local effects directly from the model's gradients, which ensures accurate estimation without resorting to artificial samples.

DALE offers a robust solution for global model interpretation, especially in high-dimensional and complex scenarios. Its computational efficiency and resistance to OOD sampling

make it a valuable tool for interpreting machine learning models. By accurately estimating feature effects, DALE helps stakeholders understand the average impact of individual features, facilitating better decision-making and trust in AI systems.

#### 2.3.2.8 Permutation Feature Importance (PFI)

Variable importance refers to the proportion of input characteristics that contribute to the output of a predictive model. Permutation Feature Importance (PFI) [42] is a method for determining the significance of features in machine learning models. This model-agnostic approach measures how much each feature's contribution can be evaluated by assessing an increase in error when its values are shuffled randomly. The PFI technique identifies which features play the most crucial role in predictions made by any given complex or simple machine learning model.

Given a model $f$ and a dataset $D$, to compute permutation importance for feature $X_j$, we have:

1. Compute the baseline performance of the model $Perf(D)$ on the original dataset.
2. Permute the values of feature $X_j$ to create a new dataset $D'$.
3. Compute the performance of the model $Perf(D')$ on the permuted dataset.
4. The importance of $X_j$ is given by the difference in performance:

$$Importance(X_j) = Perf(D') - Perf(D)$$

The steps to be considered are:

1. **Train the Model**: Train the model on the original dataset.
2. **Baseline Performance**: Measure the model's performance (e.g., accuracy, RMSE) on the original dataset.
3. **Permute Feature**: For each feature, permute its values in the dataset.
4. **Measure Performance**: Measure again but this time use shuffled datasets instead.
5. **Calculate Importance**: Compute the difference in performance between the original and permuted datasets.

As an illustration, let us consider a simple case where we want to predict house prices based on three different attributes – size(sqm), number of bedrooms and age(years). We will only look at these specific areas when it comes to evaluating their importance.

1. Suppose there exists a regression model trained specifically for predicting housing prices over some instances such as these:

- Instance 1: size = 120 sqm, bedrooms = 3, age = 10 years.
- Instance 2: size = 180 sqm, bedrooms = 4, age = 5 years.
- Instance 3: size = 240 sqm, bedrooms = 4, age = 15 years.

2. Compute the baseline performance (e.g., RMSE) of the model on the original dataset. Let's say the RMSE is 20,000.
3. For each feature, permute its values and measure the model's performance:
   - Permute ”size” and compute the RMSE. Permuted dataset: [(240, 3, 10), (120, 4, 5), (180, 4, 15)].
   - Permute ”number of bedrooms” and compute the RMSE. Permuted dataset: [(120, 4, 10), (180, 3, 5), (240, 4, 15)].
   - Permute ”age” and compute the RMSE. Permuted dataset: [(120, 3, 15), (180, 4, 10), (240, 4, 5)].
4. After permuting each feature, suppose we get the following RMSE values:
   - RMSE with ”size” permuted: 25,000.
   - RMSE with ”number of bedrooms” permuted: 22,000.
   - RMSE with ”age” permuted: 21,000.
5. Compute the importance of each feature:
   - Importance of ”size” = 25,000 - 20,000 = 5,000.
   - Importance of ”number of bedrooms” = 22,000 - 20,000 = 2,000.
   - Importance of ”age” = 21,000 - 20,000 = 1,000.
6. Calculate Importance:
   - Importance of ”size” = 25,000 - 20,000 = 5,000.
   - Importance of ”number of bedrooms” = 22,000 - 20,000 = 2,000.
   - Importance of ”age” = 21,000 - 20,000 = 1,000.

The permutation feature importance can be visualized as a bar plot where the X-axis are the feature names and the Y-axis are the importance values (change in RMSE).

#### 2.3.2.9 Break Down (BD)

Break Down (BD) [43] is an XAI method that provides local explanations for predictions made by complex machine learning models. This method decomposes a single instance's prediction into contributions from each feature. In this way, it is clear how they affect the

final output. The contributions are cumulatively calculated to show step-by-step how much each feature has impacted the prediction.

Unlike other methods, BD does not explain the overall behaviour of a model; instead it explains individual predictions one at a time. It builds up towards the final prediction by adding together features' contributions in sequence. Any type of model can be used with BD such as neural networks, tree ensembles and linear models. The order in which features are considered can influence their contributions.

Given a model $f$ and an instance $x = (x_1, x_2, \ldots, x_p)$, the Break Down method explains the prediction $f(x)$ by decomposing it into contributions from each feature. The process involves the following steps:

1. **Baseline Prediction**: A starting point which is usually average over all predictions in dataset denoted as $\hat{y}_{base}$.
2. **Cumulative Contributions**: For each feature $x_j$, compute its contribution $\Delta_j$ and add it to the cumulative prediction.

The final prediction $f(x)$ can be expressed as:

$$f(x) = \hat{y}_{base} + \sum_{j=1}^{p} \Delta_j$$

where $\Delta_j$ is the contribution of feature $x_j$.

Consider predicting house prices using three variables: size (in sqm), number of bedrooms and age(years). We shall illustrate our explanation on a single instance.

1. **Model and Data**: Assume we have built a regression model that predicts house prices based on features like the below:
   - Size: 180 sqm.
   - Number of bedrooms: 3.
   - Age: 10 years.
2. **Baseline Prediction**: Assume the baseline prediction is the mean house price, which is 300,000€.
3. **Cumulative Contributions**: Calculate sequentially the contribution of each feature. Let's assume the contributions are calculated as follows:
   - Contribution of Size:
     - Baseline prediction: 300,000€.
     - Predicted price if only size is considered: 350,000€.
     - Contribution of size: 350,000€ - 300,000€ = 50,000€.

- Contribution of Number of Bedrooms:
  - Cumulative prediction so far: 350,000€.
  - Predicted price if size and number of bedrooms are considered: 370,000€.
  - Contribution of number of bedrooms: 370,000€ - 350,000€ = 20,000€.
- Contribution of Age:
  - Cumulative prediction so far: 370,000€.
  - Predicted price if all features are considered: 360,000€.
  - Contribution of age: 360,000 - 370,000 = -10,000€.

4. **Calculate Final Prediction**:

   - Start with the baseline prediction: 300,000€.
   - Add the contribution of size: 300,000€ + 50,000€ = 350,000€.
   - Add the contribution of number of bedrooms: 350,000€ + 20,000€ = 370,000€.
   - Add the contribution of age: 370,000€ - 10,000€ = 360,000€.

   The final predicted house price is 360,000€.

### 2.3.3 Example Taxonomy

Table 2.2 has been constructed with an example taxonomy, taking into account the Section 2.3.1 regarding the taxonomy logic being applied as well as Section 2.3.2, incorporating only the XAI methods that will be integrated in MLInspect.

### 2.3.4 MLOps and XAI Tools

Explainable AI(XAI) approaches are supported by a number of MLOps tools; each has multiple functions for interpreting machine learning models. These offer functionalities with which users can easily understand and explain their model, their behaviours and decisions in advance. This could allow them to know about how their models operate in practice as well as give them a chance to spot any biases that might exist or even ensure that the ML algorithms have fairness or transparency aspects.

Data science teams and engineers have more control over creating dependable AI system since they have access to XAI within MLOps platforms. Some tools will be summarised in Sections [2.3.4.1, 2.3.4.2, 2.3.4.3, 2.3.4.4, 2.3.4.5]. Awesome Production Machine Learning repository, under Explaining black-box models and datasets section, offers more tools that integrate XAI methods.

**Table 2.2: An example taxonomy of the Popular XAI methods**

| Method | Result | Functioning | Output Format | Scope | Stage |
|---|---|---|---|---|---|
| LIME | Surrogate Models | Perturbations | Numerical, Visual, Model, Textual | Local | Post-Hoc: Model-Agnostic |
| Shapley Values | Feature Relevance | Perturbations | Numerical, Visual, Model, Textual | Local, Global | Post-Hoc: Model-Agnostic |
| Integrated Gradients | Feature Relevance | Structure Leveraging, Perturbations | Numerical, Visual, Model, Textual | Local, Global | Post-Hoc: Model-Specific |
| ALE | Feature Relevance | Perturbations | Numerical, Visual | Global | Post-Hoc: Model-Agnostic |
| DALE | Feature Relevance | Perturbations | Numerical, Visual | Global | Post-Hoc: Model-Agnostic |
| PDP | Feature Relevance | Perturbations | Numerical, Visual | Global | Post-Hoc: Model-Agnostic |
| ICE | Feature Relevance | Perturbations | Numerical, Visual | Local | Post-Hoc: Model-Agnostic |
| PFI | Feature Relevance | Perturbations | Numerical, Visual | Global | Post-Hoc: Model-Agnostic |
| Break Down | Feature Relevance | Perturbations | Numerical, Visual, Textual | Local | Post-Hoc: Model-Agnostic |

#### 2.3.4.1 InterpretML

InterpretML [25] is an open-source toolkit developed to improve the interpretability of machine learning models. It was created to aid data scientists and practitioners in order to understand, diagnose, and trust their models through interpretable methods spanning global and local perspectives.

Global interpretability methods help users understand the overall behaviour of their models. These include Explainable Boosting Machine (EBM) and Partial Dependence Plots (PDPs). EBM is a highly interpretable glass-box model that competes with state-of-the-art machine learning algorithms in accuracy while providing clear insights into feature importances and interactions. PDPs are plots that show how various features affect the model predictions by illustrating the average effect of a feature across a dataset.

On the other hand, local interpretability methods concentrate on explaining individual predictions. Some methods supported are the Shapley Additive exPlanations (SHAP) and Local Interpretable Model-agnostic Explanations (LIME). SHAP values break down a prediction into contributions from each feature, providing consistent and accurate explanations for individual predictions. LIME approximates the model locally around a specific prediction to create an interpretable explanation of the model's decision.

InterpretML maintains all the aforementioned methods as well as Morris method and Individual Conditional Expectation (ICE) Plots. Morris method is a sensitivity analysis tool that evaluates the impact of input variables on the model's output by varying one input at a time, helping to identify the most influential features. ICE plots are plots that extend PDPs by showing how different instances are affected by one feature allowing for deeper understanding of its effects.

It is easy to use as it offers a useful API for the user. Finally, it offers a comprehensive set of tools to analyse model's attributes, the performance and run what-if scenarios.

#### 2.3.4.2 Alibi

Alibi [44] is an open-source Python library providing cutting-edge explainability algorithms for machine learning models. The range of support varies from tables, data, texts, and images in different classification and regression models. Local and global scopes are covered by both model-agnostic (black-box) and model-specific (white-box) methods offered by Alibi Explain. Additionally, it can be employed in various machine learning applications.

Several key algorithms like Accumulated Local Effects (ALE), Anchor Explanations, Contrastive Explanation Method (CEM), Counterfactual Explanations or SHAP(Shapley Additive Explanations) methods are supported by Alibi Explain. These methods have diverse applications such as calculating global feature effects, finding minimum subsets of features ensuring prediction stability and generating artificial instances of the classifier's behaviour. A unified API design ensures a consistent user experience across different algorithms while at the same time, it emphasises robustness through extensive testing and

continuous integration. This ensures code correctness as well as algorithm convergence. Also, there exists a detailed documentation that offers theoretical backgrounds, usage instructions alongside end-to-end examples that render practical implementation possible. Alibi Explain integrates with deployment platforms like Seldon Core and KFServing for use in production environments.

In conclusion, Alibi Explain acts as a bridge between research on model explainability and its application to real-life industry scenarios. It has been designed to serve as a comprehensive toolkit aiding in the interpretation of machine learning models.

#### 2.3.4.3 AI Explainability 360

AI Explainability 360 [45, 46] is an open source python library designed to make AI models more transparent and interpretable. It has been developed in response to growing demands in society for AI systems, especially for the ones that offer methods to explain the reasons for their outputs, particularly in high-stakes areas such as healthcare, finance, or law. This toolkit is designed to cater for various stakeholders including data scientists, end-users and regulators with different types of explanations needed to understand AI models.

There are ten different explainer algorithms included in AIX360. Each one fits best in a different type of explanation. These algorithms help interpret model predictions, feature importance scores and data representations. Another tool in the kit is a taxonomy that aids users through the maze of explanation methods. For instance, methods can be grouped together based on whether they provide local or global explanations, if they are model-agnostic or dedicated only for certain type of models. As for the architecture of AIX360, it is built to allow easy integration of new explainability methods, providing extensibility. Its API resembles familiar machine libraries like scikit-learn making them user-friendly. All these are provided as reference in tutorials, demonstrated using web demos. These features provided by this toolbox will help many people who do not have an idea about these terms get started with understanding what AI explainability stands for.

Explainability algorithms included are Locally Interpretable Model-agnostic Explanations (LIME), Shapley Values (SHAP), Disentangled Inferred Prior Variational Auto-encoder (DIP-VAE) which helps in identifying independent, semantically significant features in the data. Another method is the Boolean Decision Rules via Column Generation (BRCG) which generates interpretable decision rules for binary classification, Contrastive Explanations Method (CEM) and ProfWeight. ProfWeight is used so that training data weights would be adjusted to improve interpretable models using complex neural network outputs. Teaching Explanations for Decisions (TED), Generalised Linear Rule Models (GLRM), ProtoDash and CEM with Monotonic Attribute Functions (CEM-MAF) are another algorithms provided.

The two metrics, Faithfulness and Monotonicity are included in AIX360 so as to evaluate the accuracy and consistency of explanation techniques. Its implementation follows AI modelling pipeline stages, making it possible to fit it into different parts, thus ensuring

seamless integration with existing workflows. Healthcare and human capital management are some of the many use cases that support its versatility in practice, thereby implying that it is not limited only by these examples.

In conclusion, AI Explainability 360 provides a range of methods and tools to increase the interpretability of AI models, addresses the needs of different stakeholders, and promotes transparent AI applications.

#### 2.3.4.4 What-if Tool (WIT)

What-if Tool (WIT) [47] is an open source application that has been developed to assist users in understanding and analysing machine learning (ML) model performance. This tool is called the What-If Tool (WIT), which enables users to interactively probe, visualise, and analyse ML systems with minimal coding and presented with hypothetical scenarios.

WIT is a versatile tool for usage across various ML frameworks since it is designed to work with any trained model and sample dataset. This tool is provided as part of TensorBoard which is a frontend for TensorFlow but also Jupyter and Colaboratory notebooks can employ it. Users are able to do "what-if" queries examining how changes in data points affect predictions given by models. Through this feature, counterfactual reasoning is supported, whereby different situations can be simulated and during these circumstances the behaviour of the models can be observed. Moreover, different visualisation options are available in WIT. These aid understanding of model behaviour among its users. The ones included are confusion matrices, histograms, scatterplots and performance metrics among others. The task of local or global model understanding can both be achieved using this tool. With WIT, also, ML fairness metrics are quantified, visualised as well as analysed by the user. By observing how models perform on different demographic groups, one can make adjustments that improve fairness according to preference. These adjustments can be integrated during computing process. One example is the intersectional analysis.

This tool is user-friendly even to non-experts who have little experience in programming. Setup requirements are minimal. Therefore, it can be widely used by all those, including, data journalists and civil society groups who wish to effectively engage with ML. WIT has many practical use cases. These include, income prediction models, age prediction models and image-based classification models. Users found out that WIT was very powerful as it helped them discover unexpected performance discrepancies and biases. It was tested through multiple rounds of user feedback, including, internal studies and external workshops. Acting as a model debugging tool, WIT was found to be powerful for performance analysis as well, making it possible to support iterative and exploratory processes.

Through interactive visualizations and a variety of different tasks for model understanding, WIT allows deep probing into one's model, thereby identifying problems which might not have been noticed before as well as taking informed decisions towards improving the model's fairness and effectiveness.

#### 2.3.4.5 DALEX

DALEX [48, 49] is a Python package designed to improve model transparency and fairness for responsible machine learning. The paper reveals the increasing opaqueness and complexity of contemporary machine learning models that come along with challenges like potential discrimination risks, non-reproducibility or decline in performance due to drift in data.

The DALEX package resolves these problems by offering an interface that combines different explainer methods and fairness approaches into a single framework which can be used for any kind of machine learning models. It builds on the previous work done on the DALEX R package but it extends its functionalities to python language which is widely used for developing machine learning algorithms. DALEX can run seamlessly with libraries like scikit-learn, tensorflow, xgboost and h2o (machine learning libraries) as well as numpy and pandas (data manipulation libraries).

DALEX includes a main explainer class. This class is a central component that serves as a uniform abstraction over different predictive models and data APIs, facilitating the computation of various explanation objects. DALEX offers both global (model-level) and local (predict-level) explanations that help us understand how a model performs overall or how a single observation behaves at a time. This can be achieved through permutation importance technique; partial dependence plots (PDP); accumulated local effects (ALE); individual conditional expectation (ICE); and Shapley values. This package, in addition, offers methods for assessing model fairness. These enable users to visualise biases or discriminations found within their machine-learning models. One can use the package to construct interactive dashboards that enable comparison of models and explore them in terms of their behaviour and fairness.

As a whole, DALEX seems like a huge achievement on the path of responsible machine learning. It provides an all-encompassing tool which integrates explainability and fairness into any machine-learning development cycle. It, thereby, makes sure that ML models are not only powerful but also transparently fair.

#### 2.3.4.6 Comparison

A comparison is depicted in Table 2.3 regarding the tools described in the previous sections. All tools can be used in a Python environment and provide a useful API that can be utilised effectively. Also, they provide an extensive documentation to review current functionality, with examples, tutorials and information about method parameterisation. InterpretML and DALEX provide their implementation as an R package, for users that do not want to replace their pipelines with the python language. What-If Tool is specifically provided to visualise what-if scenarios using a great variety of plots and it is really helpful in depicting fairness. Alibi and AIX360 are equipped with a major variety of explainability methods, but only AIX360 offers an API that can be extended more, enabling the user to use AIX360 further, implementing custom XAI methods, according to what is needed to be

**Table 2.3: MLOps & XAI Tools Comparison**

| Tool | XAI Methods | Programming Environment | Benchmarked |
|---|---|---|---|
| InterpretML | EBM, ICE, LIME, PDP, SHAP | Python, R | Yes |
| Alibi | ALE, Anchors, CEM, Counterfactuals, Integrated Gradients, PD Variance, Partial Dependence, Permutation Importance, SHAP, Similarity | Python | Yes |
| AIX360 | CEM, Disentangled Inferred Prior VAE, Individual Conditional Expectation, Integrated Gradients, LIME, ProfWeight, ProtoDash, SHAP | Python | Yes |
| WIT | Counterfactuals, PDP | Python | Yes |
| DALEX | Breakdown, LIME, Permutation Importance, SHAP | Python, R | Yes |

accomplished. DALEX is a simple tool that integrates seamlessly with machine learnings libraries and provides less XAI methods than AIX360 and Alibi.

## 2.4 Related work

We have based our work on MLInspect. Nevertheless, we have discovered that HYPPO is another tool that uses hypergraphs to represent ML pipelines. This portrayal is utilised to pick the most performant execution that minimises costs. This section will describe both MLInspect and HYPPO, giving a high-level image of how they operate.

### 2.4.1 MLInspect

It is important to ensure that data processing and machine learning (ML) pipelines are correct and efficient in the dynamic, iterative world of data science. These often consist of multiple interconnected stages such as data preprocessing, feature engineering, model training, and evaluation. Given this complexity, there are common mistakes including data leaks, distribution shifts, and unintentional data transformations which can lead to significant impact on the performance of a model.

MLInspect functions through by automatically tracing the flow of data within a pipeline by constructing an entire DAG graph that describes all the stages involved in data processing as well as model training. This graphical representation provides an opportunity to visualise every stage of activity within a pipeline for analysis purposes. Every node in the pictured DAG represents some operation or transformation while edges show how data moves between these operations. By organising it in such way MLInspect makes it easy for users to identify possible problems. In addition, automatic instrumentation of relevant function calls within a data science pipeline is one of its standout features. What this means is that MLInspect can place checkpoints as well as monitoring hooks at strategic points in the pipeline without requiring major code modifications to be made on it. By so doing, MLInspect avoids slowing down debugging and complexifying manual inspections like debug statements. MLInspect allows predefined or custom inspections to be attached to specific nodes within the DAG. Predefined inspections include detecting data leaks (where some information from test set influences training set) and distribution shifts (where statistical properties are different between test and training datasets). Besides, users can customise their own inspection rules according to their unique needs by defining custom inspections. This capability ensures that MLInspect can adapt itself for various debugging and monitoring requirements.

However, besides visualising, there are analytical components that operate on top of DAGs created by MLInspect as well. In order to trace data propagation through pipelines, MLInspect looks at these in form of flow graphs where transformations are represented as edges and vertices represent entities involved into such a process. This holistic view is invaluable when trying to find unintended biases or errors associated with any particular data transformation.

To sum up, MLInspect is a powerful tool that can be used to improve the debugging and monitoring of data science pipelines. Its ability to instrument function calls automatically and generate up an elaborate DAG representation make it a very desirable tool for data scientists and machine learning practitioners.

More of how MLInspect works will be encapsulated in Section 3.

### 2.4.2 HYPPO

HYPPO (Hypergraph-based Pipeline Optimization) is a pioneering system aimed at optimizing ML pipelines especially in exploratory scenarios requiring fast iteration cycles and

experimentation. It aims to create more efficient execution plans that take less time and cost significantly fewer computational resources by leveraging equivalences and past executions.

The novelty behind HYPPO's innovation lies in its representation of workload using directed hypergraphs. Hypergraphs, unlike conventional graphs, provide a way to capture the intricate relationships between the computations involved in ML tasks including multi-input and multi-output tasks and alternative computations called equivalences. With this advanced representation, it becomes possible for multiple computational paths to reach an end result thereby offering flexibility as well as optimisation opportunities not available with simpler graph structures. HYPPO optimises the ML pipelines by acting like the main problem is the search functionality over these directed hypergraphs. The main goal is to find an execution plan that minimises computation time while being cost-effective. The choice of artifacts to materialise forms a key part of this optimisation process. By selecting which intermediate results should be stored strategically, HYPPO maximises the reusability of such artifacts for other future calculations. Therefore, such an approach speeds up current executions plus improves efficiency during future iterations on the pipeline since these already computed results can be reused directly.

One significant contribution made by HYPPO is its handling logical equivalences within ML pipelines. Logical equivalences denote distinct implementations of a logical operation that give mutually matching outputs when employed upon similar inputs. For instance some operations may undergo using different programming languages or frameworks. Equivalences make it possible for HYPPO to discover additional opportunities for optimisations; hence substituting one implementation with another if it leads to more efficient execution. Moreover, this flexibility is valued in exploration scenarios where different methods are often being tried and revised. The system, also, keeps track of the pipeline execution, identifies possible optimisations, and applies them during the process. It records previous executions as well as their respective outputs which it utilises to guide future optimisations.

Several ML pipelines were tested using the software, revealing that it is able to obtain execution plans which are up to two orders faster or cheaper than those not optimised. These figures therefore indicate the significant cost savings HYPPO may achieve even where materialisation is not feasible. As HYPPO moves forward, it will be important to enhance its ability to handle larger and more complex pipelines, integrate with a broader range of ML frameworks, and further improve its optimisation algorithms for handling increasingly diverse and dynamic workloads.

### 2.4.3 Contribution

This thesis highlights the implementation of MLInspect. It leverages persistence of intermediate results of an ML pipeline to automate the procedure of calculating preferred XAI methods. These methods can also be shown in the DAG representation. In comparison with HYPPO, MLInspect depicts a ML pipeline using DAG, but not hypergraphs. Its purpose is to offer a better visualisation of the ML pipeline when XAI methods are used,

not to optimise the execution time. It can persist metadata between nodes but not only for the sake of optimisation but to easily calculate XAI methods without the need for re-calculations. This integration of XAI can easily support users that need to obtain valuable information from their model, providing the means to understand how the model works and therefore minimise the ML execution times. HYPPO attempts to accomplish the same job but by utilising logical equivalences. Enhanced MLInspect tries to achieve a better visualisation and debugging of an ML pipeline. It employs DAG representation, Inspections, Checks and XAI methods to help the user define the ML pipeline more accurately. HYPPO carries out a more strict purpose, to optimise the ML pipeline execution time by visualising the pipeline and maintaining intermediate steps metadata for future calculations.

# 3. THE MLINSPECT SYSTEM

MLInspect is a framework that debugs problems related to data distribution in machine learning pipelines, applying also technical bias checks. Its purpose is to eliminate bugs originating from skewed datasets which can compromise accuracy, dependability or fairness in AI systems. This involves capturing the flowcharts representing input-output relationships during data processing procedures following DAG concept so that code can be automatically instrumented with pre-defined inspections. Such tracking is lightweight because it employs common data science libraries unlike other approaches that require manual code modification. MLInspect uses standard data science libraries and does not require any changes to the code being inspected.

These are the core steps of its typical workflow:

1. **Preparation**: Find out what inspections should be set according to user's choice of desired checks.
2. **Instrumentation**: Instrument function calls within the Abstract Syntax Tree (AST) of the user's program.
3. **Execution of the Instrumented Program**: Delegate the execution of inspections to library-specific backends; execute inspections in conjunction with pipeline operations; generate the dataflow Directed Acyclic Graph (DAG).
4. **Results**: Evaluate the checks using the DAG and the inspection outcomes.

## 3.1 Concepts

The paper "MLInspect: A Data Distribution Debugging and Inspection Library for Machine Learning Pipelines" by Stefan Grafberger et al. brings out some great key ideas about how the debugging and inspection in ML pipelines is improved. Here are the main concepts included in the paper:

**Directed Acyclic Graph (DAG) Representation.** The flow of data through different preprocessing steps is represented by a directed acyclic graph (DAG) in MLInspect. In this graph, each node represents a particular operation or transformation applied to the data. This representation of dataflow is important because it allows MLInspect to look at every step systematically by tracking how the data changes along the pipeline, thereby ensuring transparency and accountability in handling of data. It can also automatically instrument code to enable inspections. The DAG helps visualising and comprehending what operations happen within pipelines.

**Annotation Propagation.** Lightweight annotation propagation is a method used in MLInspect. This entails attaching metadata about lineage and transformation events with moving pieces of input through the system pipeline; therefore, information concerning where

errors might have crept in can be obtained easily by this tool. Lineage, here, means knowing where something came from as well as all subsequent alterations that were made on it till now. Lineage tracking works by giving a unique identifier to each tuple which gets propagated by various operations such as joins or projections being done during processing stages. Thus this knowledge helps one understand every bit's flow path across various stages which may be required for purposes like debugging among others. Data truthfulness cannot only be verified but also ensured while using this approach.

**Inspections.** One feature that distinguishes MLInspect from other tools is its ability to insert checks and inspections into code automatically without manual intervention needed. Pre-designed inspections are meant for discovering frequent problems or biases encountered. These could be found during the pre-processing activities and can be data missing, data that do not match expected format or data filled with null values. This may lead to features which are easily biased. These automated checks help in preventing human mistakes and speeding up debugging process. It also maintains fairness of ML models.

Custom inspections can be implemented using the library so that individual requirements are taken care of. Statistics like histograms for sensitive groups can be computed at any point along a pipeline. Custom inspections provide flexibility for data scientists to address their specific needs.

The paper, also, introduces scan-sharing techniques which allow performing multiple inspections within one pass over the data thereby reducing overhead associated with running separate ones. It improves overall efficiency during execution time of pipeline components.

**Declarative Abstractions and Popular Data Science Libraries.** One notable thing about MLInspect is its reliance on pandas, scikit-learn, numpy and other widely used data science libraries that provide high-level declarative abstractions. Declarative programming makes it possible for users to specify what they want done without going into details on how things should be accomplished thus offering simplicity in usage even when dealing with complex tasks like large scale machine learning projects. Seamless integration into these environments ensures accessibility together with user-friendliness making this tool widely acceptable across various levels of knowledge and skills in the field of artificial intelligence (AI).

**Handling Control Flow.** It describes how the current system creates the DAG during runtime by modifying function calls and examining the stack frame. Such a method enables the library to handle complicated controls and guarantee accurate inspection outcomes.

In summary, MLInspect is an advanced and effective system for debugging problems related to data distribution in machine learning pipelines. By identifying biases automatically, it enhances the general fairness as well as quality of ML models thus becoming a must-have instrument for data scientists working on such projects.

### 3.1.1 DAG Representation

As previously mentioned, MLInspect is specialized in generating a DAG to represent a pipeline and also persist valuable data. Also, each DAG node persists an operator, which shows what function type was called (example operators defined in Table 3.1). The operator plays a crucial role in determining what kind of data to be persisted in the DAG node and how the DAG will be ultimately formed.

**Table 3.1: Example Operator Types of MLInspect**

| Function Call | Operator |
|---|---|
| `('pandas.io.parsers', 'read_csv')` | Data Source |
| `('pandas.core.frame', 'DataFrame')` | Data Source |
| `('pandas.core.frame', '__getitem__')` | Projection |
| `('pandas.core.frame', '__getitem__')` | Selection |
| `('pandas.core.frame', 'dropna')` | Selection |
| `('pandas.core.frame', 'replace')` | Projection (Mod) |
| `('pandas.core.frame', 'merge')` | Join |
| `('pandas.core.groupbygeneric', 'agg')` | Groupby/Agg |
| `('sklearn.preprocessing._encoders', 'OneHotEncoder')` | Transformer |
| `('sklearn.preprocessing._data', 'StandardScaler')` | Transformer |
| `('sklearn.tree._classes', 'DecisionTreeClassifier')` | Estimator |
| `('sklearn.model_selection._split', 'train_test_split')` | Split (Train/Test) |
| `('sklearn.preprocessing._label', 'label_binarize')` | Projection (Mod) |
| `('sklearn.pipeline', 'fit'), arg: train data` | Train Data |
| `('sklearn.pipeline', 'fit'), arg: train labels` | Train Labels |

The DAG representation, therefore, helps in visualizing and understanding the sequence of operations performed in an ML pipeline. One example is the below.

Consider the code block, which loads a dataset, merges and processes data before configuring preprocessing steps and model, then it trains and evaluates the model (Listing

3.1):

```
"""Predicting which patients are at a higher risk of complications"""

import os
import warnings

import pandas as pd
from scikeras.wrappers import KerasClassifier
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler

from example_pipelines.healthcare.healthcare_utils import (
    MyW2VTransformer,
    create_model,
)

from mlinspect.utils import get_project_root

# FutureWarning: Sklearn 0.24 made a change that breaks remainder='drop',
# that change will be fixed in an upcoming version:
# https://github.com/scikit-learn/scikit-learn/pull/19263
warnings.filterwarnings("ignore")

COUNTIES_OF_INTEREST = ["county2", "county3"]

patients = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "patients.csv",
    ),
    na_values="?",
)
histories = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "histories.csv",
    ),
    na_values="?",
)

data = patients.merge(histories, on=["ssn"])
complications = data.groupby("age_group").agg(
    mean_complications=("complications", "mean")
)
data = data.merge(complications, on=["age_group"])
data["label"] = data["complications"] > 1.2 * data["mean_complications"]
```

```
data = data[
    [
        "smoker",
        "last_name",
        "county",
        "num_children",
        "race",
        "income",
        "label",
    ]
]
data = data[data["county"].isin(COUNTIES_OF_INTEREST)]
train_data, test_data = train_test_split(data)

impute_and_one_hot_encode = Pipeline(
    [
        ("impute", SimpleImputer(strategy="most_frequent")),
        ("encode", OneHotEncoder(sparse=False, handle_unknown="ignore")),
    ]
)
featurisation = ColumnTransformer(
    transformers=[
        (
            "impute_and_one_hot_encode",
            impute_and_one_hot_encode,
            ["smoker", "county", "race"],
        ),
        ("word2vec", MyW2VTransformer(min_count=2), ["last_name"]),
        ("numeric", StandardScaler(), ["num_children", "income"]),
    ],
    remainder="drop",
)

neural_net = KerasClassifier(
    model=create_model,
    epochs=10,
    batch_size=1,
    verbose=0,
    loss="categorical_crossentropy",
)
pipeline = Pipeline([("features", featurisation), ("learner", neural_net)])

pipeline.fit(train_data, train_data["label"])
print(
    "Mean accuracy: {}"
    .format(pipeline.score(test_data, test_data["label"]))
)
```

**Listing 3.1: Example ML pipeline**

The above code block is represented in Figure 3.1 with the corresponding lines of code that are translated in their respective DAG nodes. There are also some illustrated nodes that correspond to a specific code block. For example, the code-block (Listing 3.2):

```
patients = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "patients.csv",
    ),
    na_values="?",
)
```

**Listing 3.2: Example code to DAG node depiction**

is depicted by the node with `id = 0`. The initialisation of a Dataframe is captured by using backend infrastructure. MLInspect understands that `read_csv` method is called and after an instrumentation process, it portrays the command with a DAG node. Each node persists metadata and if it has one or multiple parents. It can secure the data arguments and the non data arguments of the method. Finally, it can create edges taking into account the operator used. In the case shown, the DAG node has `DATA SOURCE` operator. More operators are included in Table 4.2.

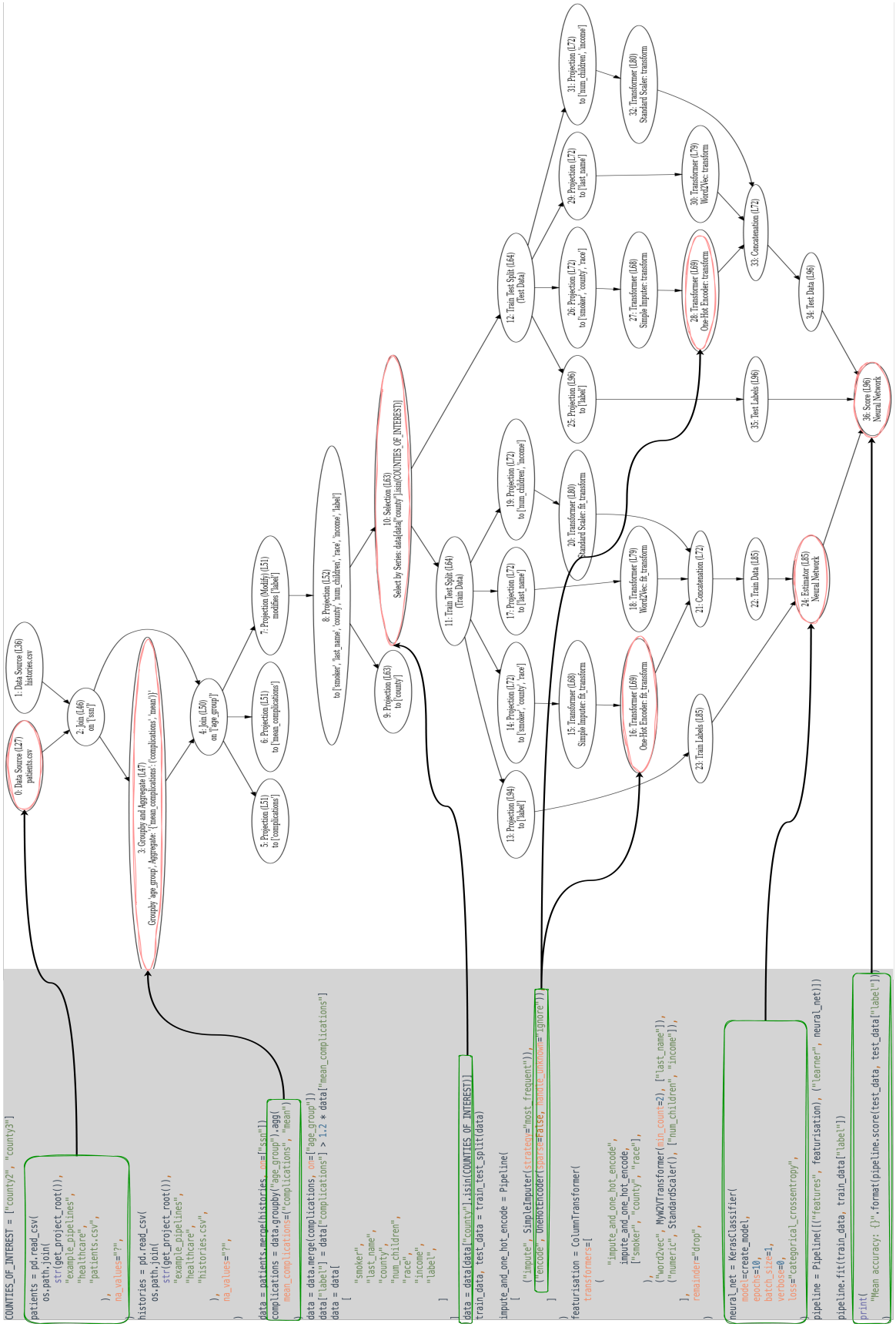



**Figure 3.1: Example DAG representation**

This DAG representation offers:

1. **Transparency**: The pipeline is easily understood at a glance, thanks to the visual representation provided by the directed acyclic graph (DAG).

2. **Debugging**: Problems can be found and fixed faster when data scientists can see the whole pipeline.

3. **Inspection**: The DAG allows for detailed checks on data transformations and model operations, which helps ensure that the pipeline is correct and robust.

4. **Documentation**: By showing how all the pieces fit together, the DAG acts as documentation for the structure of the pipeline, which aids in collaboration and knowledge sharing.

### 3.1.2 Inspections

Regarding the pipeline defined in the previous section, one can set required inspections and therefore persist additional metadata to the DAG nodes. These inspections are meant to improve understanding, transparency and reliability of machine learning systems (ML) pipelines. The inspections offered are:

- **First Row Output Inspection**: This inspection captures first row output by each operator within ML pipeline. This inspection captures the number of user input rows for each command in an ML pipeline.

- **Lineage Tracking**: MLInspect keeps track of detailed lineage for every row as they flow through the DAG of an ML pipeline. Lineage tracking lets users trace back where their data instances came from and how each row was transformed across different stages in their dataset wrangling process.

- **Histogram Computations for Sensitive Groups**: MLInspect computes histograms of operator outputs specifically tailored for sensitive groups or subsets of data. This inspection enables users to analyse and compare distributions of outputs across different protected classes.

- **Column Inspections**: MLInspect offers two inspections, one for calculating completeness of columns and another to count distinct number of columns. It also utilises one inspection that propagates sensitive columns to other operators.

- **Argument Capturing**: One simple inspection that MLInspect provides, is the one that captures the arguments passed to an operator. These are the non data ones, ie not datasets.

These inspections can be demonstrated with an example ML pipeline. The below code (Listing 3.3) shows how MLInspect's inspector can be set to use inspections. It takes as input an ML pipeline, and adds the referenced inspections. Then it gets the DAG representation and the inspection results.

```
HEALTHCARE_FILE_PY = os.path.join(
    str(get_project_root()), "example_pipelines", "healthcare", "healthcare.py"
)
inspector_result = (
    PipelineInspector.on_pipeline_from_py_file(HEALTHCARE_FILE_PY)
    .add_custom_monkey_patching_module(custom_monkeypatching)
    .add_required_inspection(RowLineage(5))
    .add_required_inspection(MaterializeFirstOutputRows(5))
    .add_required_inspection(ArgumentCapturing())
    .add_required_inspection(HistogramForColumns(["race"]))
    .add_required_inspection(CompletenessOfColumns(["race"]))
    .add_required_inspection(CountDistinctOfColumns(["race"]))
    .execute()
)

extracted_dag = inspector_result.dag
dag_node_to_inspection_results = inspector_result.
   dag_node_to_inspection_results

groupby_node = list(inspector_result.dag.nodes)[3]
inspection_results_tree = inspector_result.dag_node_to_inspection_results[
    classifier_node
]
captured_args = inspection_results_tree[ArgumentCapturing()]
```

**Listing 3.3: Setup inspection in MLInspect**

`RowLineage` inspection appends a unique identifier in each feature, a dict like `{LineageId(operator_id=1, row_id=0)}`, containing the operator id and the row id. Each row contains this kind of identifier, which can be further enriched depending on what the operator does, merging, deleting, grouping or applying other calculations to the rows. `MaterializeFirstOutputRows` prints the output of a node, but just the first number of rows defined in the argument, e.g. 5. `ArgumentCapturing` will print out the arguments defined in a node's function call, ie a command in the ML pipeline, for example, for the group by command of the healthcare script, it will capture the args: `"mean": ('complications', 'mean')`. `CountDistinctOfColumns` will return the number of distinct values in column "race". As for `HistogramForColumns` inspection, it will return a dictionary with the values of each column defined as arg, which can then be used in checks like `NoBiasIntroducedFor`, referenced in Section 3.1.3.

Overall, inspections offered by MLInspect provide complete visibility into machine learning pipelines starting from initial data transformation down to more granular levels of data lineage, quality assessment and group-specific analysis of pipeline outputs.

### 3.1.3 Checks

Checks are additional verifications that are added on top of an executed DAG when initialising the MLInspect tool. Checks may require either just the generated DAG representation or also the metadata of inspections that were implemented and integrated at the pipeline execution. Some checks provided by MLInspect tool are:

1. **NoIllegalFeatures**: Verifies that the ML pipeline does not include features that are explicitly marked as illegal or prohibited. These features are: `"race", "gender", "age"` and more can be added, according to user input.

2. **NoMissingEmbeddings**: Verifies whether or not sensitive attributes like race were included as part of input features for a given ML model. The user can specify a test minimum threshold that if they are surpassed, it outputs the features.

3. **NoBiasIntroducedFor**: Checks that the ML pipeline does not introduce bias because of operators like joins and selects. If it does, it outputs the code reference and operator responsible for the change, for the features set up by the user. The results are backed up by the distribution change that is calculated in the background, having as metadata the histogram of a feature before and after an ML pipeline command execution.

These checks can be reviewed better with an example. The below code (Listing 3.4) demonstrates how a user can execute MLInspect's inspector. It takes as input an ML pipeline, and adds the referenced checks. Then it gets the DAG representation and the check results.

```
HEALTHCARE_FILE_PY = os.path.join(
    str(get_project_root()), "example_pipelines", "healthcare", "healthcare.py"
)

inspector_result = (
    PipelineInspector.on_pipeline_from_py_file(HEALTHCARE_FILE_PY)
    .add_custom_monkey_patching_module(custom_monkeypatching)
    .add_check(NoBiasIntroducedFor(["age_group", "race"]))
    .add_check(NoIllegalFeatures())
    .add_check(NoMissingEmbeddings())
    .execute()
)

extracted_dag = inspector_result.dag
check_results = inspector_result.check_to_check_results

feature_check_result = check_results[NoIllegalFeatures()]
embedding_check_result = check_results[NoMissingEmbeddings()]

for (
    dag_node,
```

```
    missing_embeddings_info,
) in embedding_check_result.dag_node_to_missing_embeddings.items():
    print(
        "\n\033[1m{} ({})\033[0m\n{}\n{}".format(
            dag_node.operator_info.operator,
            dag_node.details.description,
            dag_node.optional_code_info.source_code,
            dag_node.optional_code_info.code_reference,
        )
    )
    print(
        "\033[1mExamples with missing embeddings: {}\033[0m".format(
            missing_embeddings_info.missing_embeddings_examples
        )
    )

no_bias_check_result = check_results[NoBiasIntroducedFor(["age_group", "race"])]

distribution_changes_overview_df = (
    NoBiasIntroducedFor.get_distribution_changes_overview_as_df(no_bias_check_result)
)
display(distribution_changes_overview_df)

dag_node_distribution_changes_list = list(
    no_bias_check_result.bias_distribution_change.items()
)
```

**Listing 3.4: MLInspect inspector and checks**

The results `NoIllegalFeatures` could be "race" and for `NoMissingEmbeddings` will be a string for each rare name with no embedding. For `NoBiasIntroducedFor`, MLInspect generates a Dataframe with the information of the code reference and which configured fields have changed above the test minimum threshold specified.

## 3.2 Architecture

The user process starts by running their pandas/sklearn-based data science pipeline with MLInspect, specifying what inspections and checks to apply. Then, relevant function calls get automatically modified by MLInspect before executing its modified version during run time. When encountering estimators/transformers during execution phase, modified function calls for these libraries direct to backend libraries that expose inputs/annotations/outputs information about operators needed for configured inspections. After that, a dataflow representation of program is generated by MLInspect and inspection results correlated with corresponding operators or checks are applied. The overall architecture of the tool can be described in Figure 3.2.

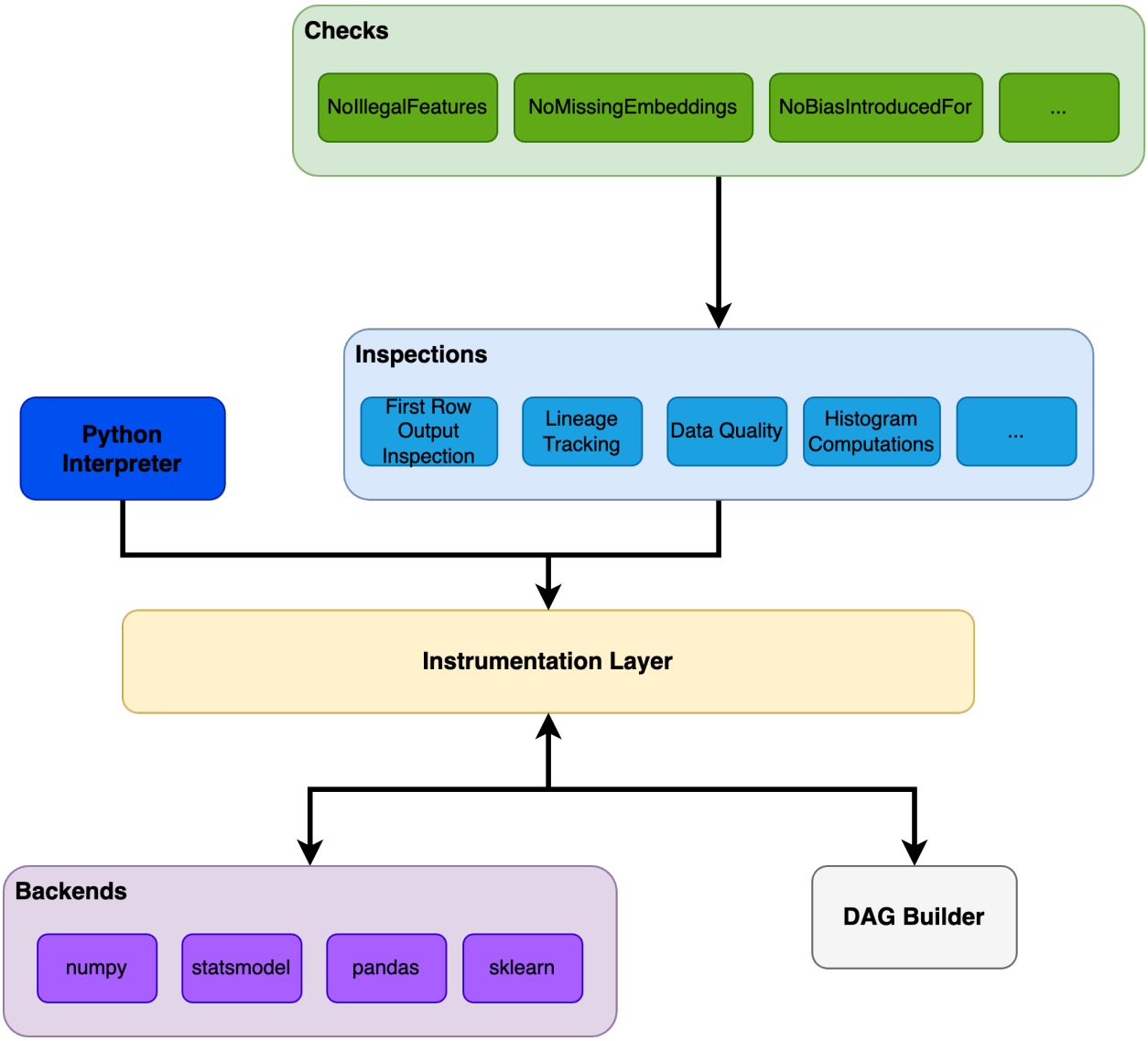



**Figure 3.2: MLInspect Architecture**

## 3.3 Implementation

Implementation is written in Python, using pip as the dependency manager. One can inspect the whole project by going to the GitHub repository. The implementation process consists of four key phases: Preparation, Instrumentation, Execution, and Results. Here's a detailed breakdown of each phase:

1. **Preparation.** In this phase, the user defines the necessary set of inspections, checks and also custom inspections and checks. This can be defined during the initialisation of the MLInspect `Pipeline Executor`.

2. **Instrumentation.** MLInspect prepares the user's program for inspection by mod-

ifying its Abstract Syntax Tree (AST). This can be achieved by instrumenting the function calls within the AST of the user's program. This step integrates hooks that will capture necessary data and metrics during execution. These hooks are implemented using `gorilla` library, which patches selected python packages, determining the input, output and metadata. These data are then used from DAG builder as well as inspections. Changes to the AST are made to ensure that the inspection mechanisms are triggered appropriately during the program's execution.

3. **Execution of the Instrumented Program.** In the third step, MLInspect runs the modified program and gathers inspection data. The execution of inspections is delegated to library-specific backends. The backends, previously defined in Section 3.2 consider both the input and output of a specific python function and using the specified operator, they delegate the appropriate information to the correct DAG representation kind of nodes. These backends are specialised, therefore, in handling the intricacies of different machine learning libraries and frameworks. Inspections are executed alongside the normal operations of the pipeline. This integration ensures that inspections are context-aware and do not hamper normal functioning of ML pipeline. During execution, the framework also constructs the DAG.

4. **Results.** The final step involves assessing the results in light of inspections outcome. Based on these, the framework evaluates the checks provided during initialisation. All required criteria are enforced in this step.

# 4. EXPLAINABLE MLINSPECT

In this section, the enhanced MLInspect that it is extended with XAI methods will be presented. The XAI methods integrated are the ones referenced in the subsections under Section 2.3.2. These are: LIME, Shapley Values, Integrated Gradients, ALE, PDP, ICE, DALE, PFI and BD. Afterwards, we will include the new inspection encapsulated into the MLInspect, the `Explainer` and further give an insight of the new architecture of the system. We will refer to other factors of the implementation and lastly we will mention any issues that arose at the time of the implementation.

## 4.1 DAG representations for XAI methods

Explainable MLInspect supports the DAG representation of the below extra packages:

- `shap` package, with `KernelExplainer` constructor and `shap_values` function that calculates the SHAP values.
- `lime` package, with `LimeTabularExplainer` constructor and `explain_instance` function.
- `sklearn_inspection` package, with `PartialDependenceDisplay` constructor and `from_estimator` function that builds either PDP or ICE plot (if `kind=individual` is set).
- `alibi` package, with `IntegratedGradients` constructor and `explain` method that calculates the corresponding explainability values. Also, `alibi` package supports `ALE` explainability method. For this reason, Explainable MLInspect supports `ALE` constructor.
- `dale` package, with `DALE` constructor and `eval` method that calculates explainability results.
- `dalex` package, with `Explainer` constructor and `model_parts` and `predict_parts` functions to calculate results.

All DAG representations that follow are based on the initial DAG representation defined in Section 3, Figure 3.1, having as base code the below listing (Listing 4.1):

```
"""Predicting which patients are at a higher risk of complications"""

import os
import warnings

import pandas as pd
from scikeras.wrappers import KerasClassifier
from sklearn.compose import ColumnTransformer
```

```
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler

from example_pipelines.healthcare.healthcare_utils import create_model_predict

from mlinspect.monkeypatching._mlinspect_ndarray import MlinspectNdarray
from mlinspect.utils import get_project_root

# FutureWarning: Sklearn 0.24 made a change that breaks remainder='drop', that change will be fixed
#  in an upcoming version: https://github.com/scikit-learn/scikit-learn/pull/19263
warnings.filterwarnings("ignore")

COUNTIES_OF_INTEREST = ["county2", "county3"]

patients = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "patients.csv",
    ),
    na_values="?",
)
histories = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "histories.csv",
    ),
    na_values="?",
)

data = patients.merge(histories, on=["ssn"])
complications = data.groupby("age_group").agg(
    mean_complications=("complications", "mean")
)
data = data.merge(complications, on=["age_group"])
data["label"] = data["complications"] > 1.2 * data["mean_complications"]
data = data[
    [
        "smoker",
        "last_name",
        "county",
        "num_children",
        "race",
        "income",
        "label",
    ]
```

```
]
data = data[data["county"].isin(COUNTIES_OF_INTEREST)]
train_data, test_data = train_test_split(data)
y_train = train_data["label"]
y_test = test_data["label"]
X_train = train_data.drop("label", axis=1)
X_test = test_data.drop("label", axis=1)

impute_and_one_hot_encode = Pipeline(
    [
        ("impute", SimpleImputer(strategy="most_frequent")),
        ("encode", OneHotEncoder(sparse=False, handle_unknown="ignore")),
    ]
)
featurisation = ColumnTransformer(
    transformers=[
        (
            "impute_and_one_hot_encode",
            impute_and_one_hot_encode,
            ["smoker", "county", "race"],
        ),
        ("numeric", StandardScaler(), ["num_children", "income"]),
    ],
    remainder="drop",
)

neural_net = KerasClassifier(
    model=create_model_predict,
    epochs=10,
    batch_size=1,
    verbose=0,
    loss="binary_crossentropy",
)
X_t_train: MlinspectNdarray = featurisation.fit_transform(train_data, y_train)
X_t_test: MlinspectNdarray = featurisation.fit_transform(X_test, y_test)
neural_net.fit(X_t_train, y_train)
print("Mean accuracy: {}".format(neural_net.score(X_t_test, y_test)))
```

**Listing 4.1: Example ML pipeline base code**

### 4.1.1 DAG representation - SHAP

The implementation is supported by information referenced in Section 2.3.2.2. `shap` package can be installed via PyPI or conda and offers extensive examples and API references for various types of models, including tabular, text, image, and genomic data. For more details, you could check the SHAP documentation.

Two functions are patched in order to get the desired DAG representation, the `KernelExplainer.__init__` and `KernelExplainer.shap_values`. `KernelExplainer` is a model-agnostic tool in SHAP (SHapley Additive exPlanations) used for interpreting predictions from any machine learning model. It approximates shapley values by sampling

from the input features' distribution. `KernelExplainer` works by creating a surrogate model using a weighted linear regression. This method is particularly useful when dealing with complex models or when the exact model infrastructure is unknown. The `shap_values` function of `KernelExplainer` computes the SHAP values for a given set of instances, which quantify the contribution of each feature to the model's prediction.This method uses a weighted linear regression to approximate the contribution of each feature to the prediction. It samples various combinations of feature values to estimate their impact. The values produced from `shap_values` can be visualised by `force_plot`[50] and `summary_plot`. The `force_plot` function creates a force plot, a visual representation which shows impact of each feature on the model's prediction for a given instance. It shows how features push the prediction from base value ie average model output to actual output.The plot has positive/negative contributions with arrows pointing towards final prediction value. The `summary_plot` function provides a comprehensive visualisation of the feature importance across a dataset. It combines feature importance with feature effects, displaying the SHAP values of features for many instances. The plot shows how much each feature contributes to the predictions and the distribution of those contributions. It mostly employs a combination of bar plots and scatter plots to show the magnitude and direction (positive or negative impact) of feature contributions.

The below code that calculates the shapley values is appended to the main ML pipeline referred in Listing 4.1.

```
import shap

shap.initjs()
explainer = shap.KernelExplainer(neural_net.predict, X_t_train)
shap_values = explainer.shap_values(X_t_test[:2], nsamples=100)
shap.force_plot(
    explainer.expected_value,
    shap_values,
    X_t_test[:2],
    feature_names=featurisation.get_feature_names_out(),
)
shap.summary_plot(
    shap_values,
    X_t_test[:1],
    feature_names=featurisation.get_feature_names_out(),
    plot_type="bar",
)
```

**Listing 4.2: Calculate shapley values**

the new DAG representation is shown in Figure 4.1. The newly added nodes can be shown with blue formation. The same visualisation inspection will be applied across all XAI methods that will be mentioned in the next sections.

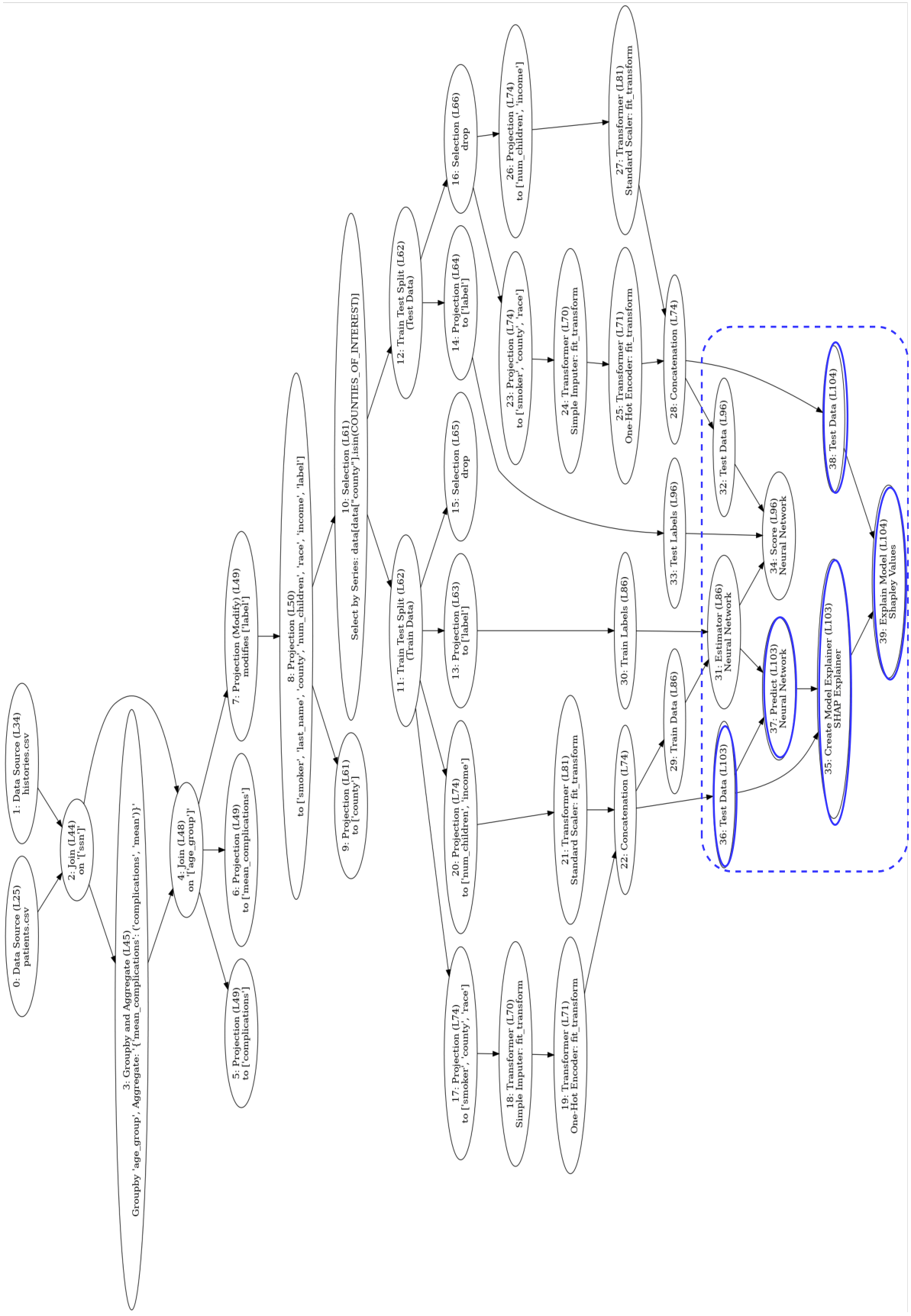



**Figure 4.1: DAG representation with Shapley Values**

What is added to the existing DAG representation are two new nodes, one for the initialisation of the explainability object and a new one to calculate the actual explainability values. This new DAG representation can be also described using the partial diagram in Figure 4.2. The diagram shows that the explainer, in order to be initialised, needs the

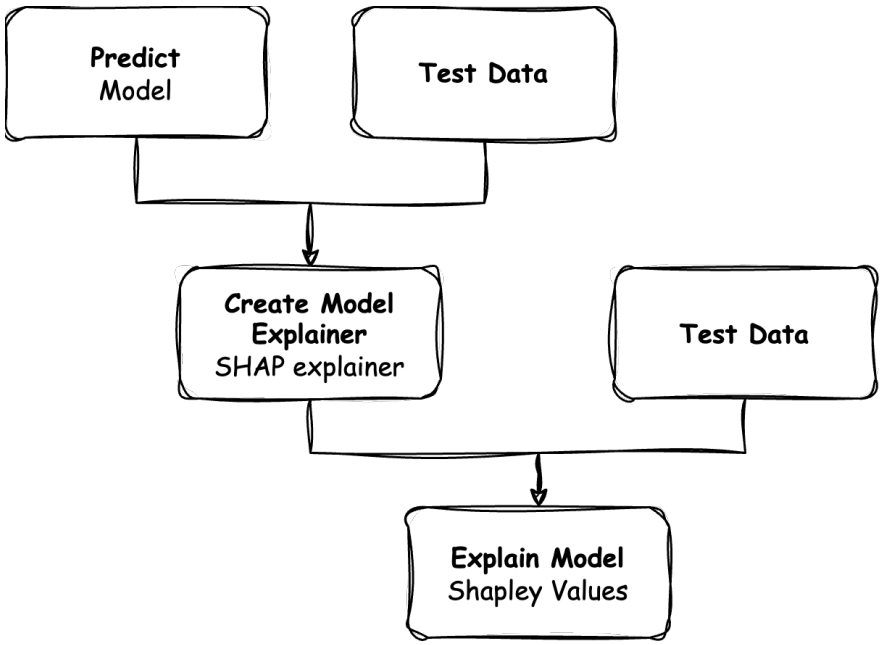



**Figure 4.2: DAG representation with Shapley Values - Partial**

predict function of the model as well as test data. Then, using this explainer and new test data, shapley values can be calculated.

### 4.1.2 DAG representation - LIME

Background information for how LIME works can be found under Section 2.3.2.1. PyPi `lime` package has been used to integrate the XAI method. The `LimeTabularExplainer` is a class in the LIME package that can be used for interpreting tabular data models. The working principle is slightly different as it perturbs the data and fits a simpler or more interpretable model like Linear Regression around an instance. This helps users to know which features drive the model's prediction about this particular instance. Datasets with categorical and continuous features are bravely supported by the `LimeTabularExplainer` because it explains complex model predictions with simple, local explanations. The `explain_instance` function generates an explanation for a given instance of dataset using the `LimeTabularExplainer` class. It perturbs the instance by sampling around it, creating a new dataset of similar instances. It, then, collects the model's predictions for these perturbed instances. It fits an interpretable model to these instances, using their distances from the original instance as weights. Finally, it identifies the influence of each feature on the prediction for the specific instance. This approach results in a locally interpretable, totally understandable and enough to reveal what leads to the predictions.

Using the below code-block (Listing 4.3), one can calculate the explainability results of LIME XAI method. This was also appended in the main ML pipeline defined in Listing 4.1.

```
import lime.lime_tabular

explainer = lime.lime_tabular.LimeTabularExplainer(
    X_t_train,
    mode="classification",
    feature_names=featurisation.get_feature_names_out(),
    class_names=[False, True],
)
result = explainer.explain_instance(X_t_test[0], neural_net.predict_proba)
result.show_in_notebook()
```

**Listing 4.3: Calculate LIME**

This produces the DAG representation in Figure 4.3.

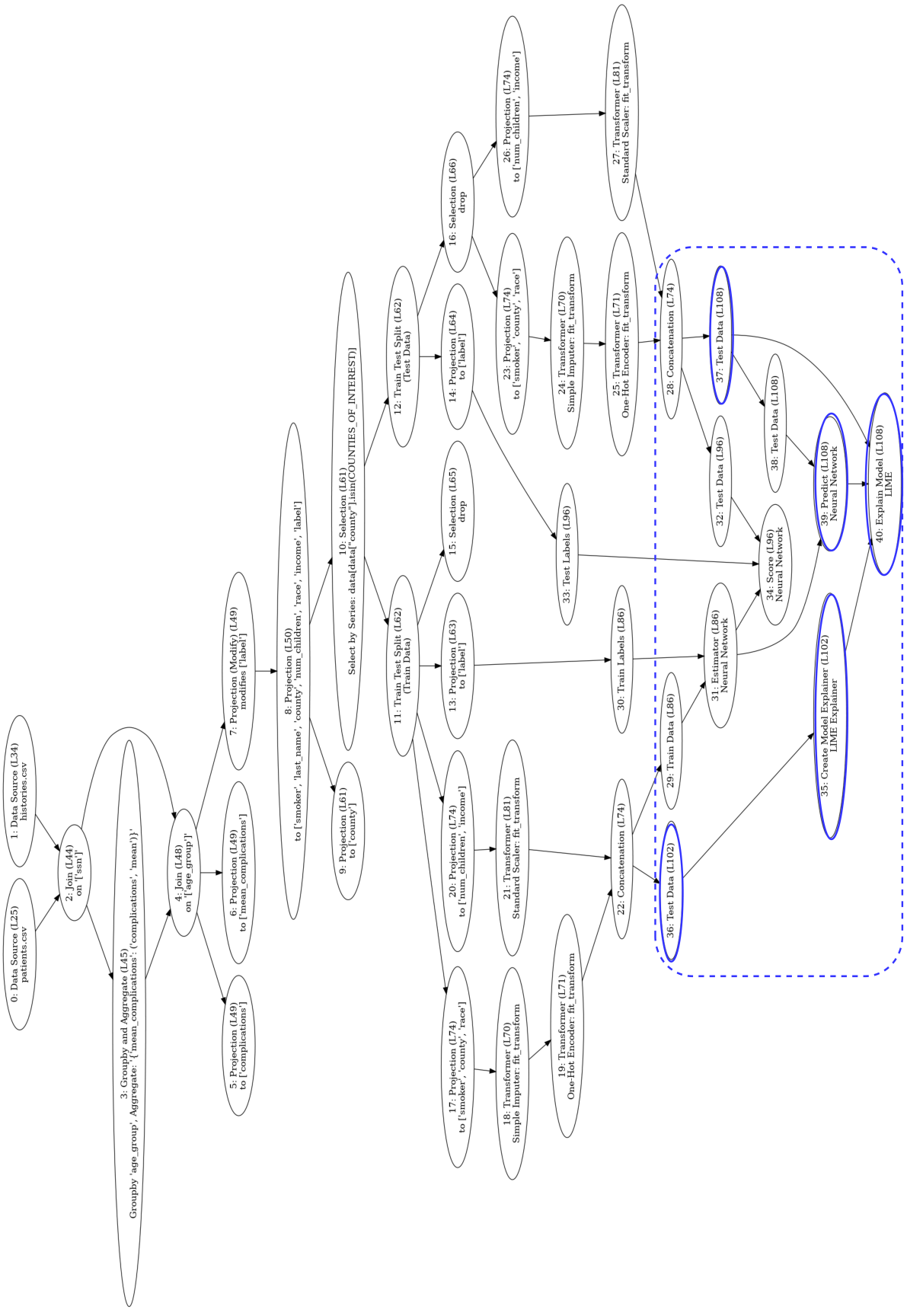



**Figure 4.3: DAG representation with LIME**

What is added is two new nodes for the initialisation of the explainability object and a new one to calculate the actual explainability values. This can be also described using the partial diagram in Figure 4.4. The explainer takes as arguments the model and the test

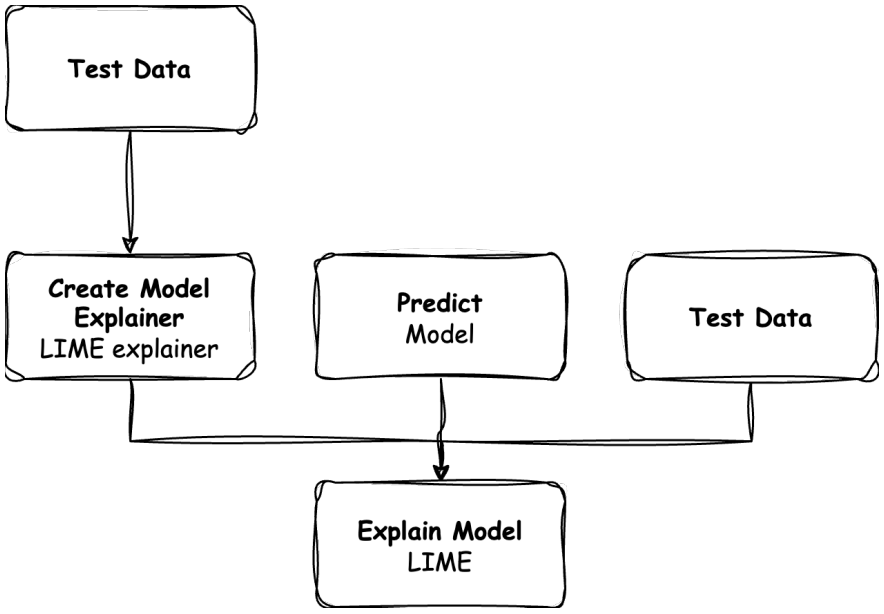



**Figure 4.4: DAG representation with LIME - Partial**

data. It needs the predict function of the model to calculate the results.

### 4.1.3 DAG representation - PDP & ICE

In this section, it will be demonstrated how PDP and ICE XAI methods were integrated in the DAG representation of MLInspect. For how PDP works, one can check Section 2.3.2.5.

Partial Dependence Plots (PDPs) provide an insight into how individual feature or features influence a machine learning model's output. To show such relationship, each plot focuses on only one feature at a time thereby visualising the impact that feature has on the prediction made.

To see which selected functionalities partial dependence of a machine learning model relies on, `scikit-learn` includes `PartialDependenceDisplay` class for straightforward visualisation and explanation, using function `from_estimator`.

The same function can be utilised to visualize an ICE curve, taking advantage the `kind` parameter. ICE plot is described thoroughly in Section 2.3.2.6.

So, adding the PDP and ICE code-blocks at the end of the Listing 4.1:

```
1 from sklearn.inspection import PartialDependenceDisplay
2
3 # pdp
```

```
display_pdp = PartialDependenceDisplay.from_estimator(
    estimator=neural_net, X=X_t_train, features=[1, 2], kind="average"
)

# ice
display_ice = PartialDependenceDisplay.from_estimator(
    estimator=neural_net, X=X_t_train, features=[1, 2], kind="individual"
)
```

**Listing 4.4: Calculate PDP and ICE**

The DAG representations can be described in Figures 4.5 and 4.6, for PDP and ICE respectively.

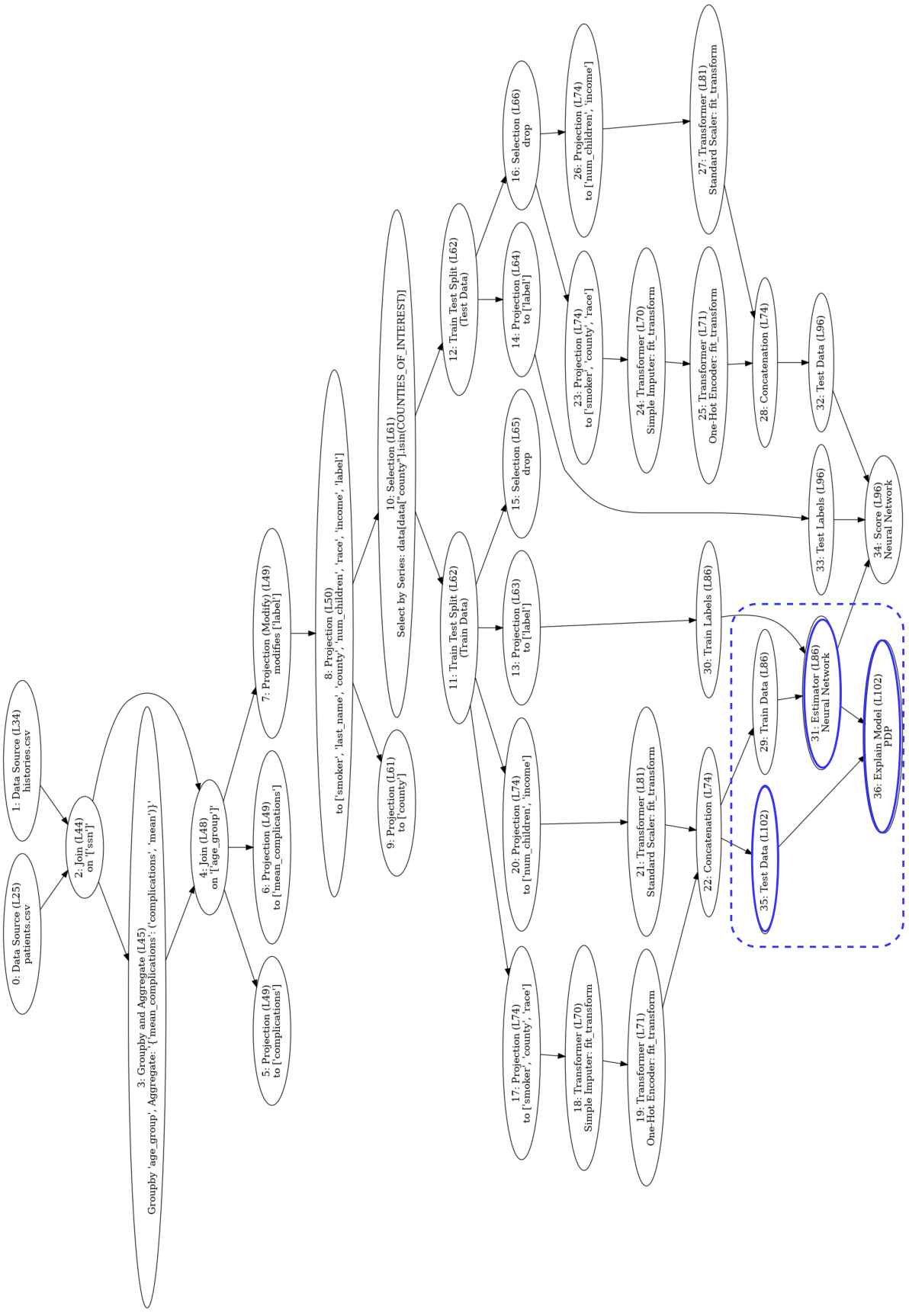



**Figure 4.5: DAG representation with PDP**

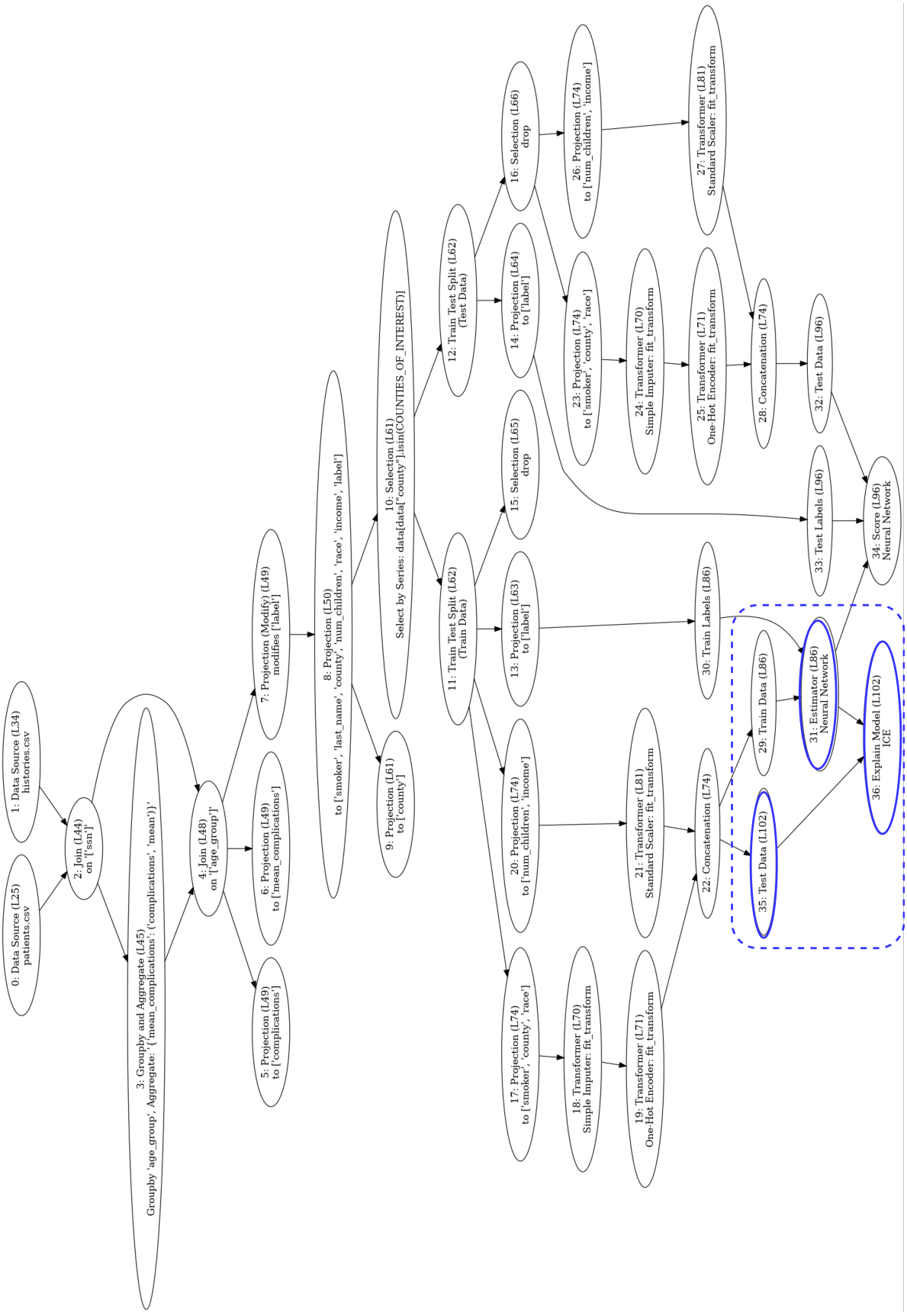
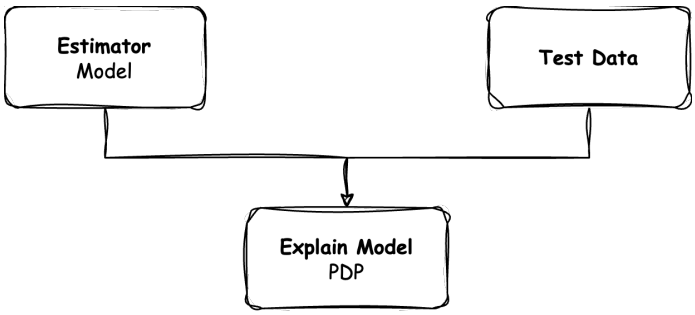
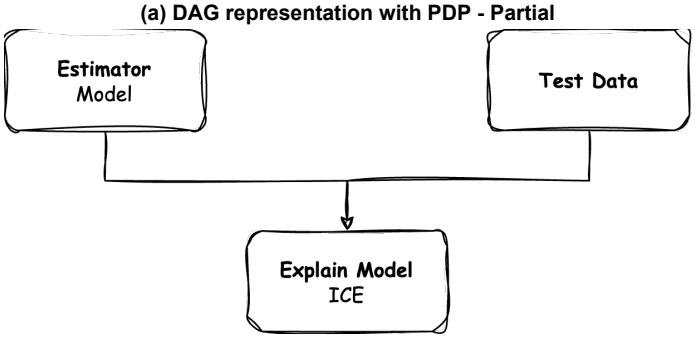



**Figure 4.6: DAG representation with ICE**

And the partial diagrams defining what nodes were added to the existing DAG representation, can be shown in Figures 4.7a and 4.7b, for PDP and ICE respectively.




(a) DAG representation with PDP - Partial




(b) DAG representation with ICE - Partial

The package needs only the estimator and test data to produce the plots, be it PDP or ICE.

### 4.1.4 DAG representation - ALE

ALE has been presented thoroughly in Section 2.3.2.4. In this section, it will be specified what PyPi package was used and the implementation under it.

For calculating ALE and then representing it in DAG nodes, `alibi` package was used. By means of `ALE.explain` function, `ALE` constructor sets up an explainer that can measure the explainability of a feature. Within the initialisation phase for this class, there is need to input all parameters into the constructor including the model prediction function as well as names of features it will take as arguments with. The test data is necessary for interpretation purposes by the `explain` method. The `explain` method takes as input a dataset (in this case `X_t_train`) and computes the ALE for each feature across the data. The main results refer to how every individual predictor contributes to prediction on average; these are quantified by ALEs . An output from this function would consist of an explanation object containing calculated values for each variable such as age or sex. The explanation can then be plotted using the `plot_ale` method of `alibi` package.

The below listing (Listing 4.5) is added to the end of the Listing 4.1.

```
1 from alibi.explainers import ALE, plot_ale
```

```
2
3 ale_explainer = ALE(
4     neural_net.predict_proba,
5     feature_names=featurisation.get_feature_names_out(),
6     target_names=[False, True],
7 )
8 explanation = ale_explainer.explain(X_t_train)
9 plot_ale(explanation)
```

**Listing 4.5: Calculate ALE**

The DAG generated is presented in Figure 4.8.

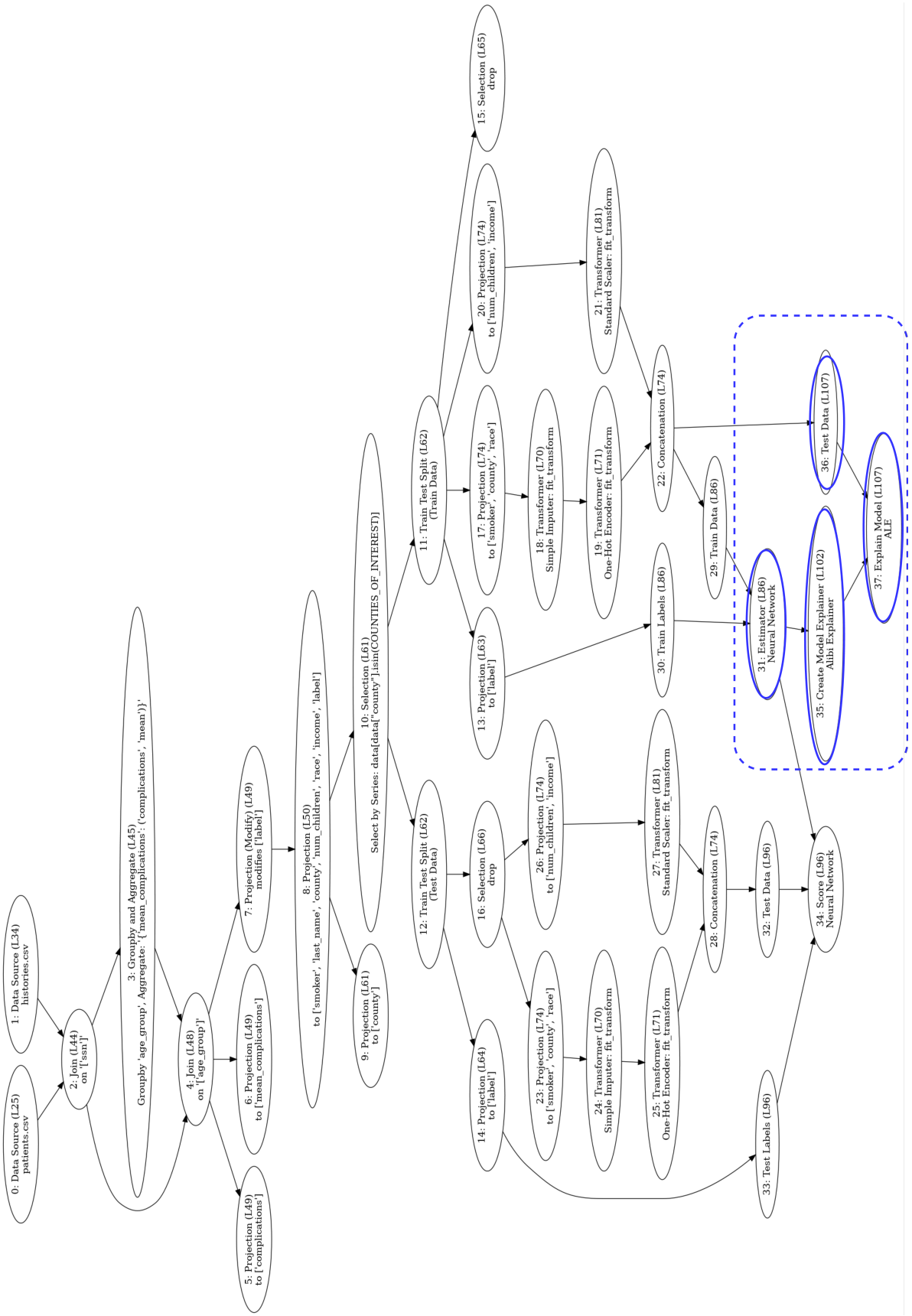



**Figure 4.8: DAG representation with ALE**

And the partial one in Figure 4.9. ALE explainer needs the model to be fitted and test data

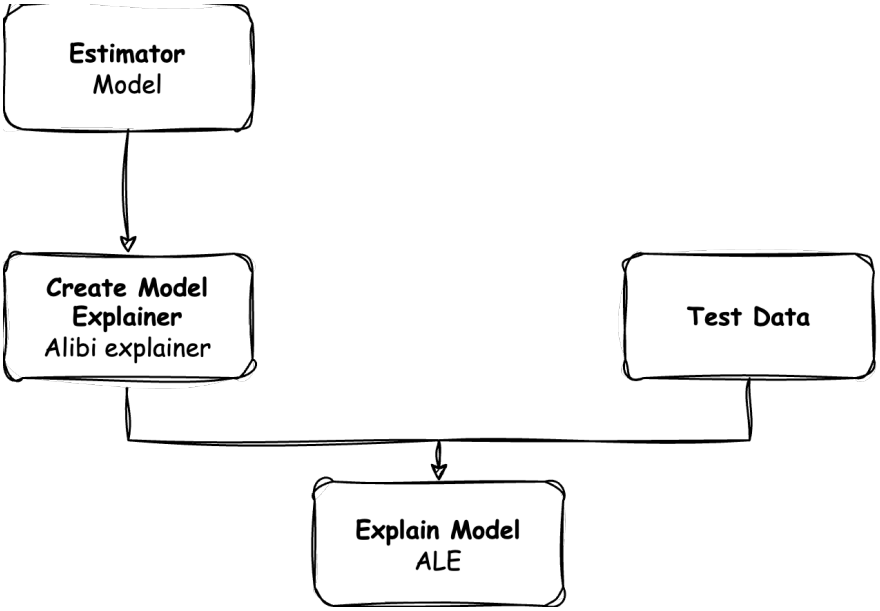



**Figure 4.9: DAG representation with ALE - partial**

to compute the explanation.

### 4.1.5 DAG representation - Integrated Gradients

Integrated Gradients XAI method has been comprehensively defined in Section 2.3.2.3. The `alibi` implementation of the integrated gradients method is specific to TensorFlow and Keras models. It needs a constructor `IntegratedGradients` and then an `explain` method to be called from the `IntegratedGradients` object so as to create an explanation object. This object contains the explained instances results.

As described in the above sections, the XAI method calculation is added as a last step in Listing 4.1.

```
from alibi.explainers import IntegratedGradients

ig = IntegratedGradients(
    model=neural_net.model_,
    method="gausslegendre",
    n_steps=50,
    internal_batch_size=100,
)
explanation = ig.explain(X=X_t_test[:1], baselines=None, target=0)
```

**Listing 4.6: Calculate Integrated Gradients**

The full code produces the DAG displayed in Figure 4.10.

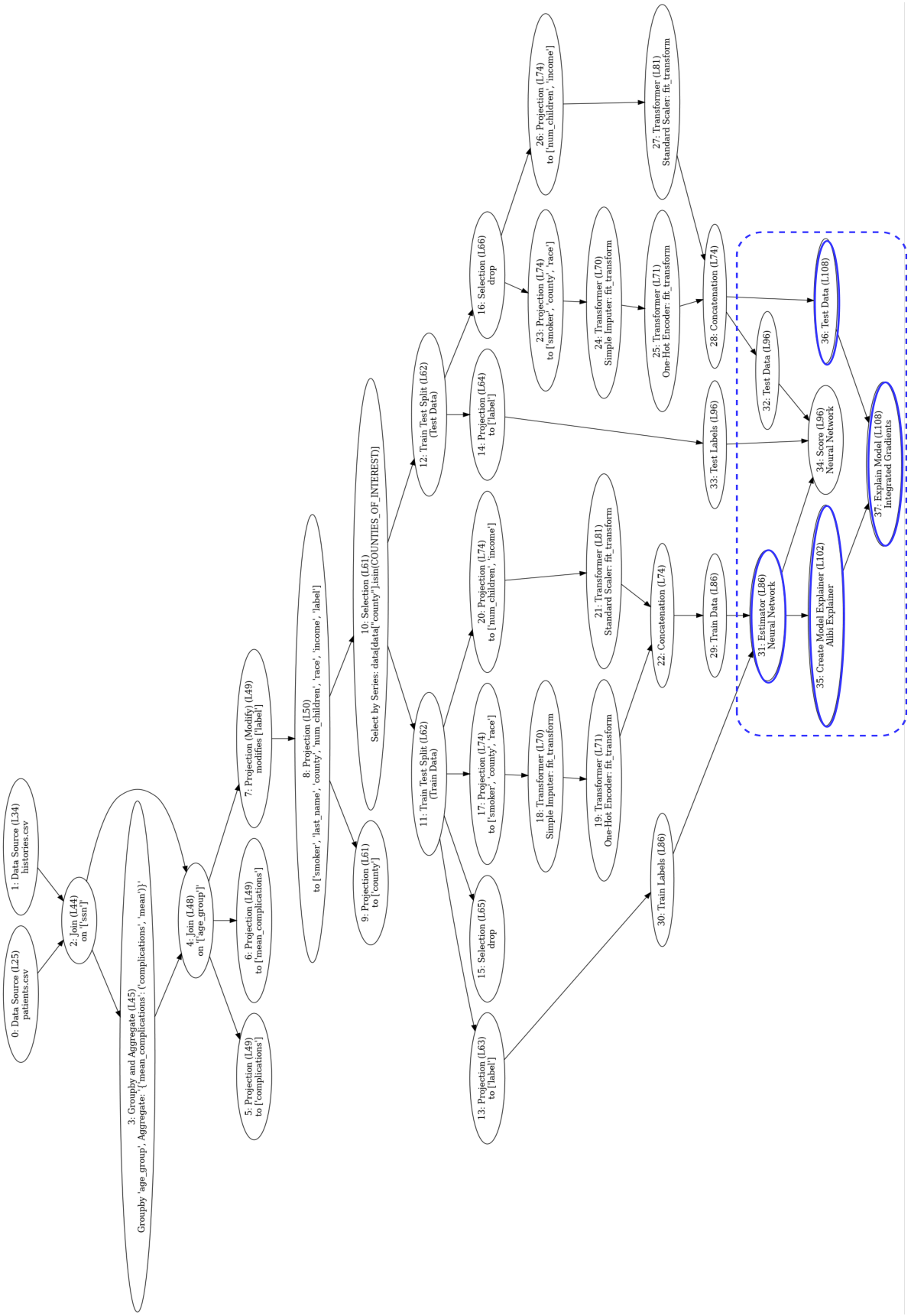



**Figure 4.10: DAG representation with Integrated Gradients**

The partial DAG showing which nodes were added can be shown in Figure 4.11. IG

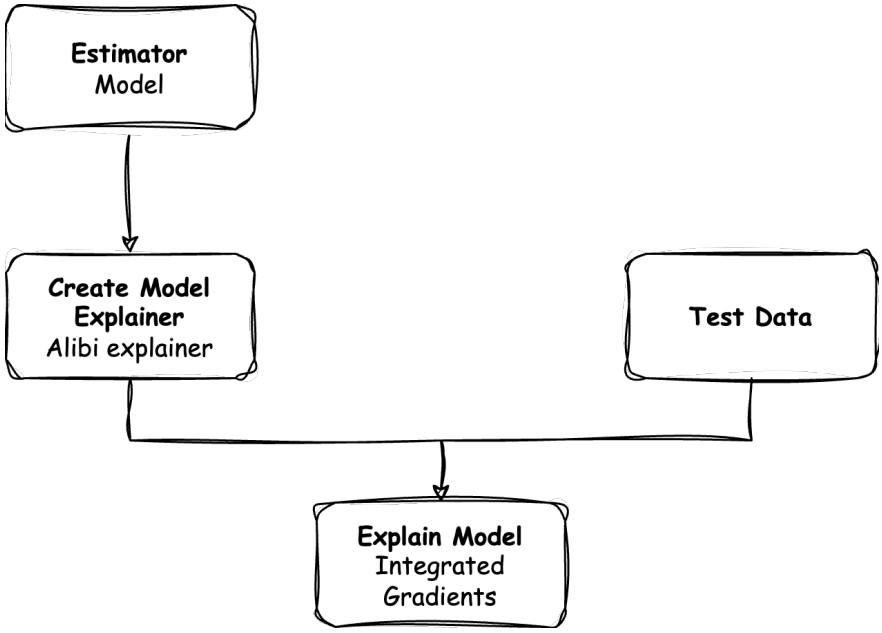



**Figure 4.11: DAG representation with Integrated Gradients - partial**

explainer needs the model to be fitted and test data to compute the explanation, just like for the ALE case.

### 4.1.6 DAG representation - DALE

More about DALE can be read in Section 2.3.2.7. In order to apply DALE, we consulted its implementation defined in the GitHub Repository. `DALE.__init__` and `eval` are patched to create the appropriate nodes in the DAG. `DALE.__init__` is the constructor of the class, which initialises the DALE object with the data, the model, and the Jacobian of the model. The `_dale_func` method is a static method that returns the DALE function for a specific feature. This function computes the accumulated effect of the feature. The `eval` method of DALE class is responsible for evaluating the DALE function on the data provided. It accepts two arguments: `x`, which is a numpy array of data, and `s`, which is an integer that represents the feature index number. The specified feature of interest denoted by the `s` variable will will be evaluated using the whole `x` dataset. Therefore, its aim is to compute how much cumulative impact each given feature had on a dataset and return this information.

As will all other XAI cases, the below Listing 4.7 is appended at the end of the base code defined in Listing 4.1.

```
1 from features.explainability.dale.dale import DALE
2
3 def model_grad(inp):
4     x_inp = tf.cast(inp, tf.float32)
```

```
    with tf.GradientTape() as tape:
        tape.watch(x_inp)
        preds = neural_net.model_(x_inp)
    grads = tape.gradient(preds, x_inp)
    return grads.numpy()

dale = DALE(data=X_t_train, model=neural_net, model_jac=model_grad)
dale.fit()
explanations = dale.eval(x=X_t_test, s=0)
```

**Listing 4.7: Calculate DALE**

This generates the DAG shown in Figure 4.12.

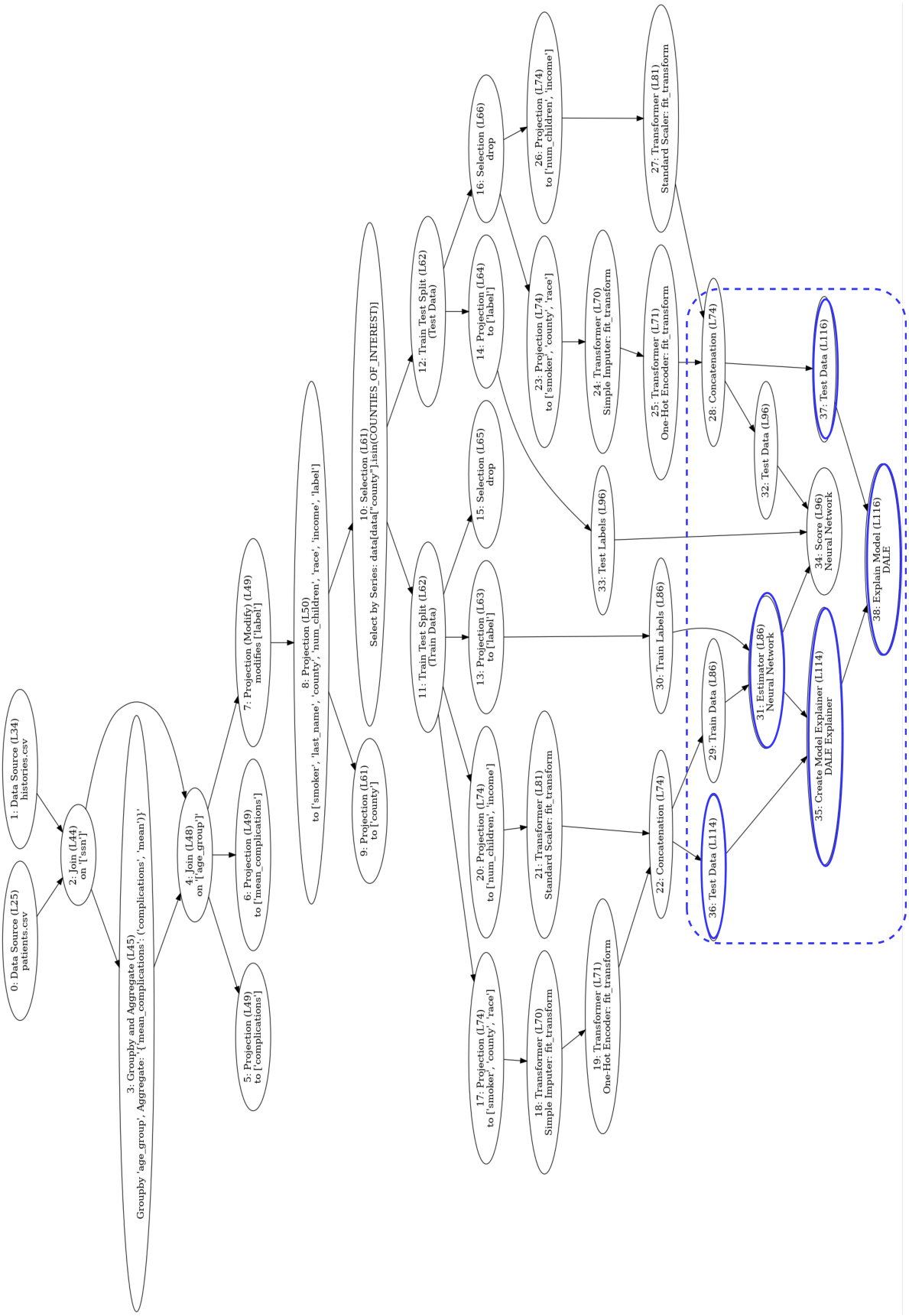



**Figure 4.12: DAG representation with DALE**

The partial DAG can be found in Figure 4.13. In the above diagram, it is described better

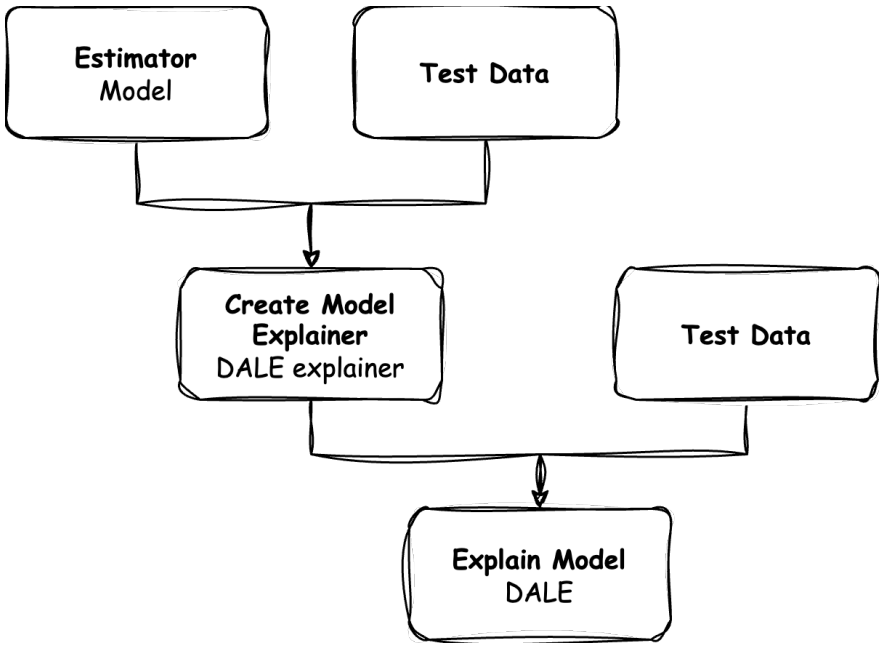



**Figure 4.13: DAG representation with DALE - partial**

which nodes are created and the input of each node for explainability calculation. The explainer of DALE needs as input the DALE constructor as well as test data. The DALE constructor should have access to the estimator and test data to calculate the explainer's metadata.

### 4.1.7 DAG representation - PFI and BD

Variable importance is a measure of what proportion each input characteristic (variable) is accountable for in determining the output of a predictive model. There are visualisation tools used in interpretable machine learning to explain the prediction of a model at an individual level. These plots aim to decompose the model's prediction for a specific instance into contributions from each input feature. Breakdown plots focus on explaining the prediction of a machine learning model for a single instance rather than providing a global model explanation.

`dalex` package that is employed in Explainable MLInspect supports Shapley values, ALE, PFI and BD plots. In this implementation, it is only utilised for the last two mentioned, ie PFI and BD plots. More about these two can be seen in Sections 2.3.2.8 and 2.3.2.9 respectively. To start with, `dalex` explainer is created with initial train data, training labels and the model to be fit. `model_parts` method computes Permutation Feature Importance (PFI), which is a global interpretability measure useful to understand the importance of different features in trained models. `predict_parts` function helps to compute the prediction breakdown for one observation so as to understand how each feature affects final predictions. `predict_parts` method works by calculating the contribution of each feature to the

prediction for a given observation; it does this by changing the value of each feature one by one and observing how that affects prediction.The contributions are then aggregated into detailed breakdowns of predictions.

The below code is attached at the end of Listing 4.1.

```
import dalex

dalex_explainer = dalex.Explainer(
    neural_net, X_t_train, y_train, predict_function=KerasClassifier.predict
)
explanation = dalex_explainer.model_parts()
train_explanation = explanation.result
df = pd.DataFrame([X_t_test.view(np.ndarray)[0]], index=["first_row"])
test_explanation = dalex_explainer.predict_parts(df, label=df.index[0]).result
```

**Listing 4.8: Calculate PFI and BD plots**

This generates the DAG in Figure 4.14.

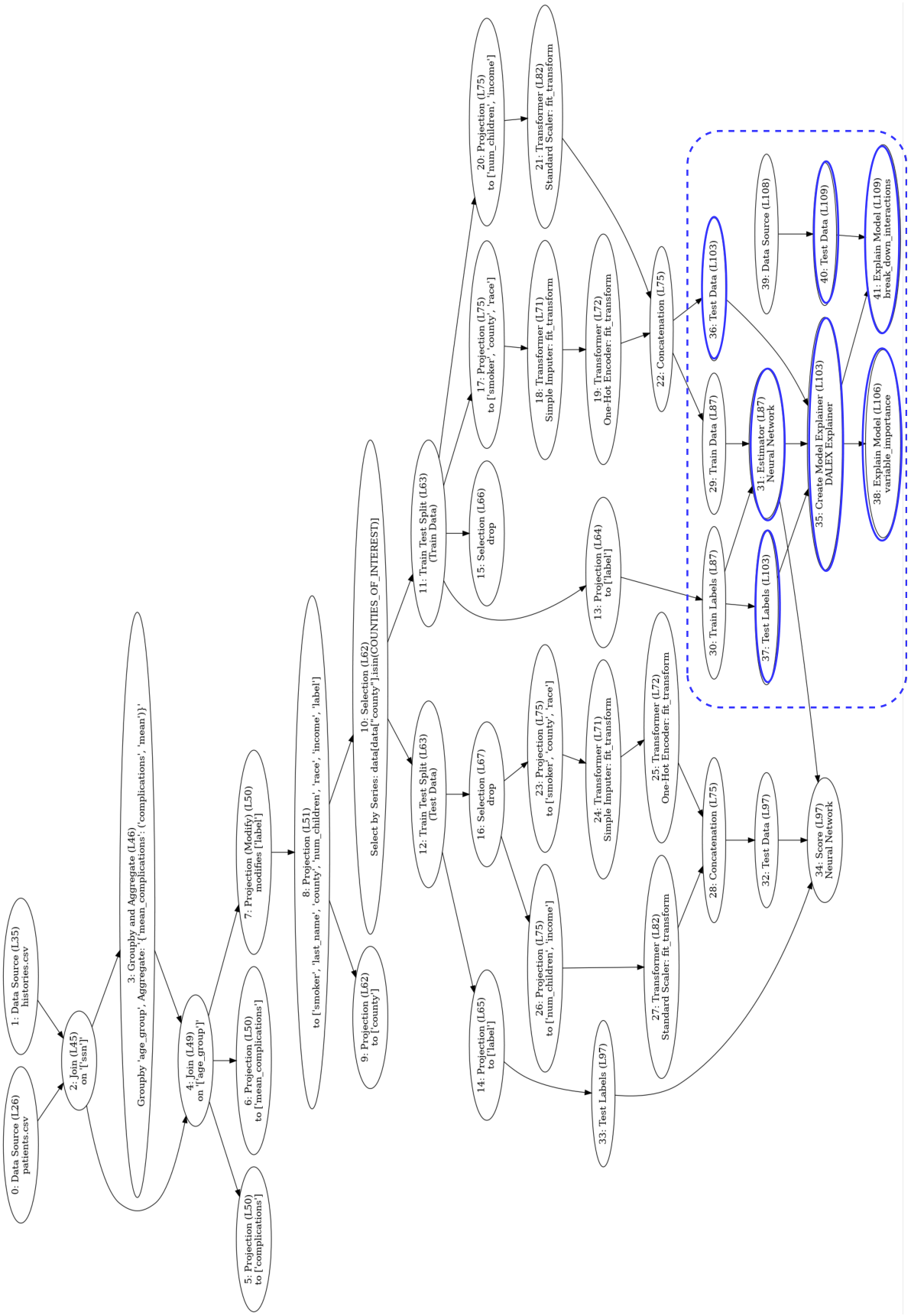



**Figure 4.14: DAG representation with BD/PFI**

And the partial DAG showing which nodes were added can be found in Figure 4.15. PFI

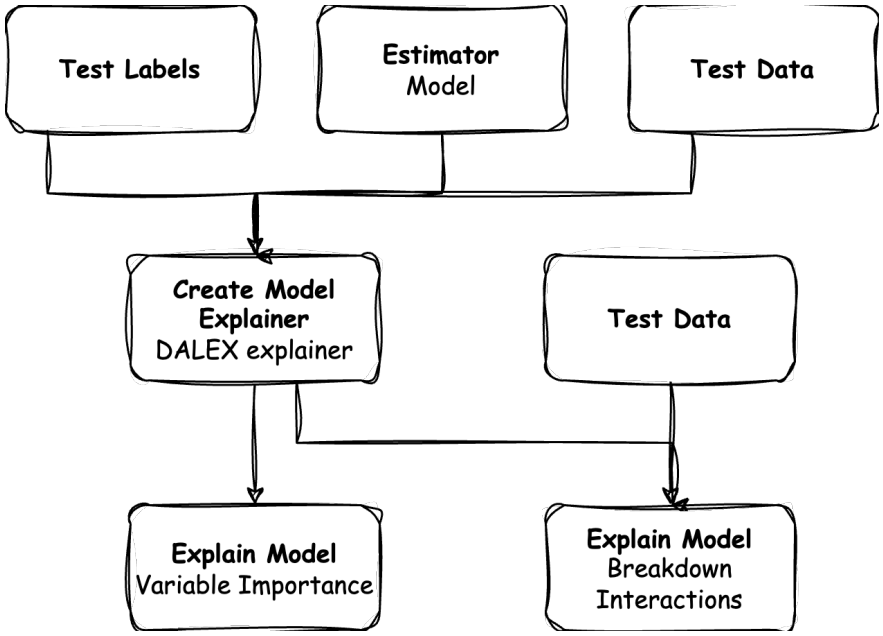



**Figure 4.15: DAG representation with BD/PFI - partial**

and BD plots can be calculated by giving a test dataset as well as the `dalex` explainer. The explainer needs the test labels, test data and a model estimator.

## 4.2 Explainer Inspection

The provided Python code in this sections is part of the Explainable MLInspect that aims to explain machine learning models. It uses a combination of different explainability methods, which are integrated in the `Explainer` class.

`Explainer` class is an inspection class that takes a list of explainability methods, test data, feature names and testing labels as its arguments. This class has a `visit_operator` method called. If an operator is visited during this inspection, it will pass through this method.

The class constructor is defined in Listing 4.9:

```
class Explainer(Inspection):
    def __init__(
        self,
        methods: List[ExplainabilityMethodsEnum],
        test_data: Any,
        feature_names: Optional[List[str]],
        test_labels: Optional[List[Any]],
    ) -> None:
        ...
```

**Listing 4.9: Explainer inspection constructor**

If this operator is an estimator and supported for explainer inspection by the explainer inspection class; when certain explainability methods are selected by the explainer instance, it applies them on the given estimator. The output of these explainer methods will be stored in `_results` attribute of `Explainer` object. The explainability methods supported are defined in Table 4.1. There is a separate if-statement for each explana-

**Table 4.1: Supported Explainability Methods Enumeration**

| Name | Value |
|---|---|
| SHAP | Shapley Values |
| LIME | Lime |
| PDP | Partial Dependence Plots |
| ICE | Individual Conditional Expectation |
| INTEGRATED_GRADIENTS | Integrated Gradients |
| ALE | Accumulated Local Effects |
| DALE | Differential Accumulated Local Effects |
| DALEX | Descriptive Machine Learning Explanations |

tion method in `visit_operator` function in `KernelExplainer`. For instance, if SHAP is chosen then library SHAP would be imported, `KernelExplainer` created and SHAP values computed for test data (Listing 4.10).

```
def visit_operator(
    self,
    inspection_input: Union[
        InspectionInputDataSource,
        InspectionInputUnaryOperator,
        InspectionInputNAryOperator,
        InspectionInputSinkOperator,
    ],
) -> Iterable[Any]:
    ...
if self._operator_type == OperatorType.ESTIMATOR:
    ...
    if model:
        if ExplainabilityMethodsEnum.SHAP in self.methods:
            ...
        if ExplainabilityMethodsEnum.LIME in self.methods:
            ...
        if ExplainabilityMethodsEnum.PDP in self.methods:
            ...
        if ExplainabilityMethodsEnum.ICE in self.methods:
            ...
        if (
            ExplainabilityMethodsEnum.INTEGRATED_GRADIENTS in self.methods
            and is_neural_network(model)
        ):
```

```
            ...
        if ExplainabilityMethodsEnum.ALE in self.methods:
            ...
        if ExplainabilityMethodsEnum.DALE in self.methods and
    is_neural_network(model):
            ...
        if ExplainabilityMethodsEnum.DALEX in self.methods:
            ...
```

**Listing 4.10: Explainer visit operator method**

The SHAP results then are persisted in a dictionary with key `ExplainabilityMethodsEnum.SHAP` in `_results` attribute. The same happens with all other explanation techniques. For the methods wanting one instance or one feature, the first one is selected by default. The implementation persists the data under key defined in Table 4.1 name field (Listing 4.11). Explainability is implemented by following Section 4.1.

```
if ExplainabilityMethodsEnum.SHAP in self.methods:
    import shap

    explainer = shap.KernelExplainer(model.predict, train_data)
    results = explainer.shap_values(self.test_data[:2])
    self._results[ExplainabilityMethodsEnum.SHAP] = {
        "explainer": explainer,
        "results": results,
    }
```

**Listing 4.11: Example Explainer results setup**

The `get_operator_annotation_after_visit` method will be executed after visiting the operator. This method returns results from the explanation functions and also resets `_operator_type` and `_results` attributes (Listing 4.12).

```
def get_operator_annotation_after_visit(self) -> Any:
    assert self._operator_type
    if self._operator_type is OperatorType.ESTIMATOR:
        result = self._results
        self._operator_type = None
        self._results = {}
        return result
    self._operator_type = None
    self._results = {}
    return None
```

**Listing 4.12: Reset Explainer results**

Explainability results are stored in an estimator DAG node. One example of usage is the `inspection.ipynb` under `examples/shap` package. Loading and pre-processing data occupies initial part of the Jupyter notebook. Reading patient and history data from CSV files, merging them together and carrying out some transformations form the first steps before narrowing it down to specific counties within interest area alone. The `train_test_split` function is used to split the data into training and test sets (Listing 4.13).

```
data = patients.merge(histories, on=['ssn'])
...
data = data[data['county'].isin(COUNTIES_OF_INTEREST)]
train_data, test_data = train_test_split(data)
```

**Listing 4.13: Example data train/test split**

This code later defines a preprocessing pipeline to deal with missing values through imputation and categorical variable encoding. Also, it scales numerical features. Then this pipeline is applied to the training and testing data (Listing 4.14).

```
impute_and_one_hot_encode = Pipeline(
    [
        ("impute", SimpleImputer(strategy="most_frequent")),
        ("encode", OneHotEncoder(sparse=False, handle_unknown="ignore")),
    ]
)
...
X_t_train: MlinspectNdarray = featurisation.fit_transform(train_data, y_train)
X_t_test: MlinspectNdarray = featurisation.fit_transform(X_test, y_test)
```

**Listing 4.14: Example data preprocessing**

Then MLInspect library's `PipelineInspector` class is used to inspect a machine learning pipeline in the notebook. `Explainer` inspection is specified for the `PipelineInspector` which applies the SHAP as its explainability method on the model of that pipeline (Listing 4.15).

```
inspector_result = (
    PipelineInspector.on_pipeline_from_py_file(EXAMPLE_PIPELINE)
    .add_required_inspection(
        Explainer(
            [ExplainabilityMethodsEnum.SHAP],
            X_t_test.view(np.ndarray),
            featurisation.get_feature_names_out(),
            [False, True],
        )
    )
    .execute()
)
```

**Listing 4.15: PipelineInspector with Explainer setup**

The results of the inspection are then printed out. The notebook specifically prints the results for the estimator operator in the pipeline (Listing 4.16).

```
relevant_node = [
    node
    for node in extracted_dag.nodes
    if node.operator_info.operator
    in {
        OperatorType.ESTIMATOR,
    }
][0]
```

```
...
inspection_result = dag_node_to_inspection_results[relevant_node][
    Explainer(
        [ExplainabilityMethodsEnum.SHAP],
        X_t_test.view(np.ndarray),
        featurisation.get_feature_names_out(),
        [False, True],
    )
]
print(inspection_result)
```

**Listing 4.16: Print Explainer inspection results**

Finally, `shap` library is used to create a force plot, a summary plot and a decision plot for SHAP values. These plots give graphical insight into how model generates its predictions (Listing 4.17).

```
shap.force_plot(
    explainer.expected_value,
    shap_values,
    X_t_test[:2],
    feature_names=featurisation.get_feature_names_out(),
)
...
shap.summary_plot(
    shap_values,
    X_t_test[:1],
    feature_names=featurisation.get_feature_names_out(),
    plot_type="bar",
)
...
shap.decision_plot(
    explainer.expected_value,
    shap_values,
    X_t_test[:1],
    feature_names=featurisation.get_feature_names_out(),
    link="logit",
)
```

**Listing 4.17: Visualize shapley values**

In summary, the `inspection.ipynb` notebook demonstrates how one can make use of both `mlinspect` and `shap` libraries to inspect a machine learning pipeline and explain predictions made by the model.

## 4.3 Architecture

The new overall architecture of the tool can be described by the diagram in Figure 4.16.

The diagram outlines the architecture of an explainable machine learning inspection system within a Docker [51] container. The added inspection is the `Explainer` which was

referenced in Section 4.2. A python interpreter exists to execute inspection processes and interacting with the system. Also, an instrumentation layer can be found, which acts as a middleware that helps communication between the inspections, checks, backend and DAG builder. The packages are the ones already defined in MLInspect Section 3.2. Regarding python packages, the ones added are:

- `shap`
- `lime`
- `alibi`
- `sklearn_inspection`
- `dale`
- `dalex`

DAG Builder has been enhanced with constructing the nodes of explainability, shown in Section 4.1. These are the `Create Model Explainer` and `Explain Model`. The project is also integrated with Make [52] commands to easily setup the Docker container and startup a Jupyter notebook server[53]. More will be concluded in Section 4.4. `Make-file` and `make` commands provide a user interface that allows users to interact with the Docker container. The workflow is:

1. `Checks` are performed to ensure data integrity and absence of bias.
2. `Inspections` are executed, interacting with the Python interpreter.
3. The `Instrumentation Layer` connects these inspections with the backend libraries and the DAG Builder.
4. The `Backends` provide the computational and analytical power needed for inspections.
5. The `DAG Builder` constructs the execution flow.
6. `Make` prompts interface facilitates user interaction and prompts generation.

## 4.4 Implementation

To improve the project's capabilities, a new package that focused on explainability has been integrated. It introduces advanced tools and methodologies aimed at offering insights into how machine learning models behave and make decisions. This includes samples for the new DAG representation, a new inspection defined in Section 4.2 and a new backend implementation `ExplainabilityBackend`.

The `ExplainabilityBackend` class has two main methods: `before_call` and `after_call`.

The method `before_call` accepts two arguments: `operator_context` and `input_infos`. `operator_context` is an instance of `OperatorContext` which contains information about the operation that was performed just before calling the back-end. `input_infos` is a list of `AnnotatedDfObject` instances, which are Dataframes with additional metadata. This function returns the `input_infos` variable as it receives it (Listing 4.18).

```
def before_call(
    self,
    operator_context: OperatorContext,
    input_infos: List[AnnotatedDfObject],
) -> List[AnnotatedDfObject]:
    return input_infos
```

**Listing 4.18: ExplainabilityBackend before_call method**

The `after_call` method is more complex. It takes four parameters: `operator_context`, `input_infos`, `return_value`, and `non_data_function_args`. The `return_value` is the result of the operation, and `non_data_function_args` is a dictionary or any other object that contains non-data arguments for the function (Listing 4.19).

```
def after_call(
    self,
    operator_context: OperatorContext,
    input_infos: List[AnnotatedDfObject],
    return_value: Any,
    non_data_function_args: MappingProxyType | Any = MappingProxyType({}),
) -> BackendResult:
```

**Listing 4.19: ExplainabilityBackend after_call method**

Inside the `after_call` method, there's a conditional statement that checks the type of the operator in the `operator_context`. If the operator is `OperatorType.EXPLAINABILITY` or `OperatorType.CREATE_EXPLAINER`, it calls the `execute_inspection_visits_nary_op` function with the provided parameters. This function performs an explanation or a creation of an explainer operation regarding the data provided, if the source that called it is either an explainability method or a creation of an explainer respectively (Listing 4.20).

```
if operator_context.operator == OperatorType.EXPLAINABILITY:
    return_value_be = execute_inspection_visits_nary_op(
        operator_context,
        input_infos,
        return_value,
        non_data_function_args,
    )
 elif operator_context.operator == OperatorType.CREATE_EXPLAINER:
    return_value_be = execute_inspection_visits_nary_op(
        operator_context,
        input_infos,
        return_value,
        non_data_function_args,
```

```
14      )
```

**Listing 4.20: ExplainabilityBackend handle operator**

The operator types supported are defined in Table 4.2.

**Table 4.2: Supported Operator Types**

| Name | Value |
|---|---|
| DATA_SOURCE | Data Source |
| MISSING_OP | Encountered unsupported operation! Fallback: Data Source |
| SELECTION | Selection |
| PROJECTION | Projection |
| PROJECTION_MODIFY | Projection (Modify) |
| TRANSFORMER | Transformer |
| CONCATENATION | Concatenation |
| ESTIMATOR | Estimator |
| SCORE | Score |
| PREDICT | Predict |
| TRAIN_DATA | Train Data |
| TRAIN_LABELS | Train Labels |
| TEST_DATA | Test Data |
| TEST_LABELS | Test Labels |
| JOIN | Join |
| GROUP_BY_AGG | Groupby and Aggregate |
| TRAIN_TEST_SPLIT | Train Test Split |
| CREATE_EXPLAINER | Create Model Explainer |
| EXPLAINABILITY | Explain Model |

If the operator does not belong to any of these expected types, then it raises a `NotImplementedError` exception with an associated message indicating that `ExplainabilityBackend` doesn't support given operation type (Listing 4.21).

```
1 else:
2     raise NotImplementedError(
3         "ExplainabilityBackend doesn't know any operations of type '{}' yet!".
    format(
4             operator_context.operator
5         )
6 )
```

**Listing 4.21: ExplainabilityBackend unhandled operator**

Finally, the `after_call` method returns the result of the `execute_inspection_visits_nary_op` function or raises an error if the operator type is not supported. This back-

end is capable of persisting in the explainer nodes, valuable metadata of the explainability methods integrated.

To make it easier to understand and visualise data processing workflows, examples and reviews were included in Directed Acyclic Graph (DAG) representations. These visualisations help users understand their workflow pipelines better by showing how data changes along different stages. They are well demonstrated in Section 4.1.

The novel Differential Accumulated Local Effects (DALE) approach was integrated to estimate feature effects in machine learning models. For this reason, some parts of DALE codebase were copied and adjusted to suit the project framework.

In terms of utils used, new `make` commands were added to bring out the project's user interface. These commands make it easier to launch and interact with the GUI of this project, enabling users to easily access and use its numerous features and tools. As such, a new Makefile was created which would help automate some repetitive tasks as well as streamline the development process so that the project could be built in seconds instead of minutes like before. In addition, these commands included also running tests or documenting code by executing specific rules. Regarding the tools used, code quality and consistency standards were also made to be enforced by pre-commit hooks [54] before any changes will be committed into the repository. These are hooks that automatically check for issues like code formatting, syntax errors and coding guidelines compliance thus making sure that the project's code base is always clean and maintainable. In order to streamline dependency management and ensure consistent environments across different development and deployment stages, Docker and Poetry [55] were used. Docker container was configured in order to maintain cache and expose the Jupyter host. This guarantees that the project will run similarly regardless of the host system; it also makes scaling very convenient plus facilitates easy deployments. Poetry was employed for managing python dependencies. Dependency resolution is simplified with poetry, package management and environment isolation making it possible to maintain and update project's dependencies easily. Its alternatives include traditional tools like pip and virtualenv. To ensure high code quality as well as coverage, code coverage was implemented. This measures how much tests have covered a piece of code. This enables to find out untested components in the codebase thus increasing overall test coverage. Furthermore, GitHub Actions were used to set up a Continuous Integration/Continuous Deployment (CI/CD) pipeline that does the following: automatically builds the project Docker image, runs tests, and deploying updates whenever there are changes being pushed into repository so that at any given time software could remain deployable.

A technique called monkey patching is used to change or extend the behaviour of certain libraries dynamically [56]. The implementation in `/features/explainability/monkey_patching` module must be followed if an end-user wants to enable more libraries in this monkey patching process. For example, patching SHAP includes the `KernelExplainer.__init__` and the `KernelExplainer.shap_values` methods, which are going to be patched by a module named `patch_shap.py`. One method creates a node for the explainer and the other for the actual explanation being computed in a user pipeline. Other modules are also included in the aforementioned path, for patching LIME, PDP, ICE,

ALE, IG, DALE, PFI and BD plots.

Finally, creation and maintenance of wide-ranging test pipelines was done to test the functionality and reliability of the components that make up the project. Such pipelines take into account different aspects of the system such as unit tests and integration tests. All these are put in place so as to ensure all parts function together correctly and efficiently.

## 4.5 Issues

The complexity of the intricacies of different packages made it difficult to manage dependencies. Upgrading dependencies often caused new issues, especially involving libraries such as `scikeras` and `gensim`. These problems emanated from version incompatibilities and changes in the APIs or internal functioning of these libraries. The structure of dependency tree became complicated and ensuring compatibility across all packages required a lot of effort and troubleshooting. Our Docker environment, for instance, posed a variety of challenges when setting up Jupyter. Specifically, Jupyter had to be able to work within Docker Networking configurations on Docker containers. A number of concerns such as connectivity issues, port mapping, and guaranteeing persistent data storage required attention for a stable and operational setup to be achieved. Flakiness was exhibiting by our test suite, which means that tests would fail at times while sometimes the passed, even if we did not change any code at all. This inconsistency created issues since it undermined the trustworthiness of our testing process. Unreliable results were due to flaky tests as they are indicators of other problems like race conditions, external system dependencies or wrong test isolation that had to be identified and fixed.

There were further complications with the predict method of the estimators which necessitated additional monkey-patching methods which were not initially part of the project. Debugging can be difficult when there are maintenance issues associated with monkey patching even though it might sometimes be necessary. Correctly implementing patches without affecting other parts of the codebase was essential here. We had to check whether we should apply monkey patching by considering syntax in our pipeline configurations. This step ensured that patches were applied selectively as well as correctly because it maintained pipeline integrity while introducing necessary modifications into it accordingly. The integration also involved copying pieces from the official package DALE (Differential Accumulated Local Effects) into ours. This needed additional adaptation thus becoming complex for us given its non-official nature. A careful manual procedure was adopted during this integration process to avoid any bug introduction and to effectively onboard DALE functionalities.

Since 2021, the `gorilla` package on PyPI has not had a new version release which poses significant problems for projects relying on it. Therefore, as Python changes or expands, `gorilla` is liable of being incompatible with subsequent versions of Python or other libraries widely used in data science and machine learning workflows.

In summary, our project faced multiple challenges related to dependency management,

environment setup, flaky tests, method patching, and integration of non-standard packages. Each of these issues required specific strategies and solutions to ensure a smooth and functional development process. Addressing these challenges was crucial to maintaining the robustness and reliability of our machine learning workflows.

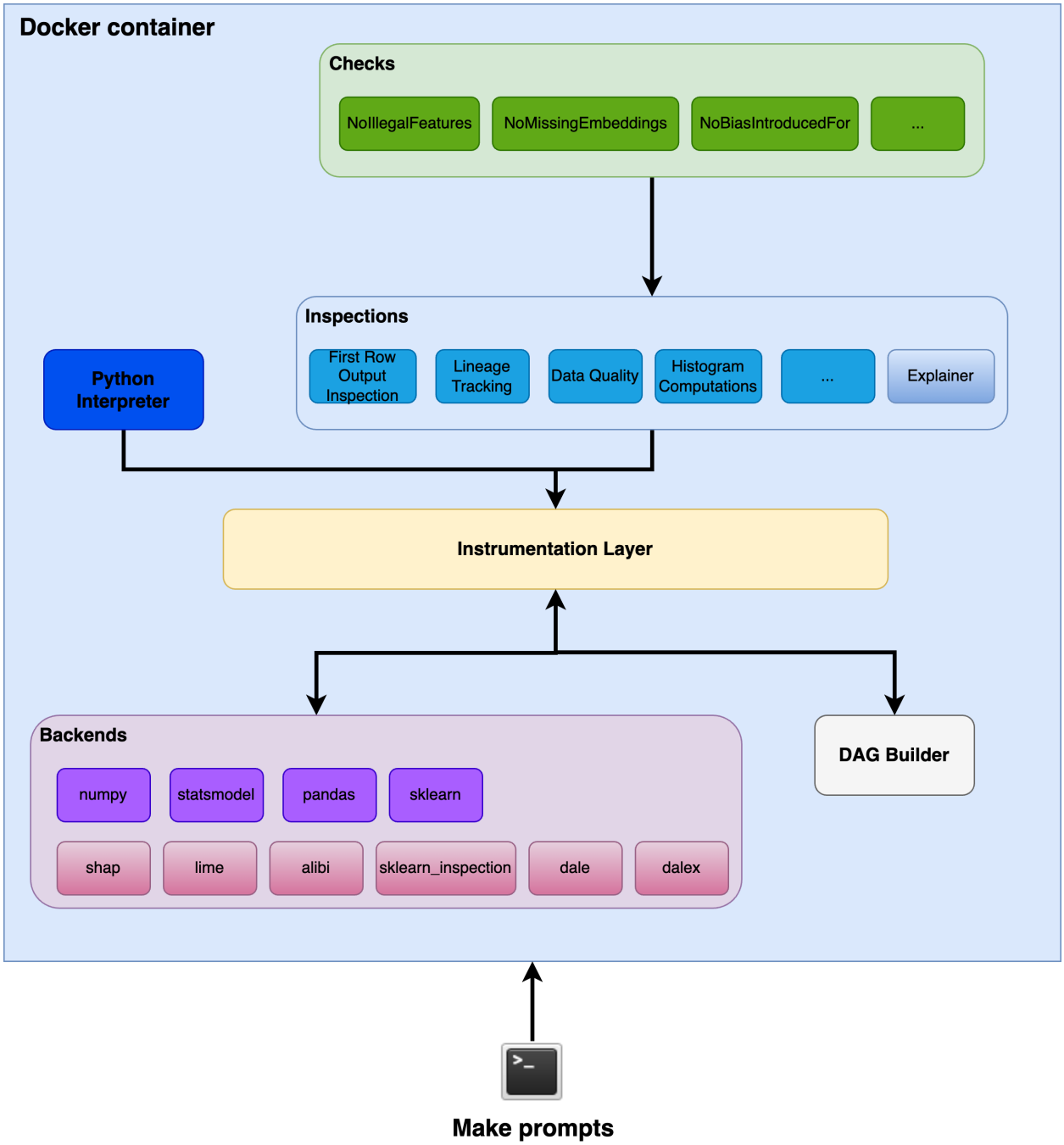



**Figure 4.16: Explainable MLInspect Architecture**

# 5. DEMONSTRATION, SCENARIOS AND EVALUATION

To illustrate how the MLInspect framework works with the addition of explainability feature, a set of user stories will be presented, to convey better how a user can interact with the project. These user stories will show the impact of the Explainable MLInspect in using and setting up explainability methods to run on the background, when a pipeline is executed.

Let's say that a data scientist wants to understand better the model in a pipeline, to gain insight as to how each feature of the data set impacts the model accuracy. Let's also hypothesise that this data scientist's name is Dr. Antonia, having a Ph.D. in data science and working in a healthcare industry. Dr. Antonia aims to develop a predictive model that identifies patients at high risk of complications after surgery. She needs to ensure that the model is not only accurate but also interpretable.

**Inspections Scenario**. This scenario encompasses the majority of situations where one might use techniques to explain a model:

1. The data scientist should install MLInspect, executing:

```
make run
```

**Listing 5.1: Makefile run command**

   She should only have installed prior to executing the tool, `docker` and `make` commands and be inside the source folder of the project. Now, as described in Section 4.4, a container is initialised, containing all the dependencies in a virtual environment, named e.g. `venv`.

2. Then, she gets the jupyter notebook link (e.g. session link) from the previous command's output and pastes it inside a browser.

3. She creates a notebook file, setting also the kernel of the notebook to previously referred `venv`.

4. After initialisation, Dr. Antonia starts by gathering patient data, including demographic information, medical history, and other patient details, an example code can be the below (Listing 5.2):

```
COUNTIES_OF_INTEREST = ["county2", "county3"]

patients = pd.read_csv(
    os.path.join(
        str(get_project_root()), "example_pipelines", "healthcare", "patients.csv"
    ),
    na_values="?",
)
histories = pd.read_csv(
    os.path.join(
```

```
        str(get_project_root()), "example_pipelines", "healthcare", "
    histories.csv"
    ),
    na_values="?",
)

data = patients.merge(histories, on=["ssn"])
complications = data.groupby("age_group").agg(
    mean_complications=("complications", "mean")
)
data = data.merge(complications, on=["age_group"])
data["label"] = data["complications"] > 1.2 * data["mean_complications"]
data = data[
    ["smoker", "last_name", "county", "num_children", "race", "income", "
    label"]
]
data = data[data["county"].isin(COUNTIES_OF_INTEREST)]
train_data, test_data = train_test_split(data)
y_train = train_data["label"]
y_test = test_data["label"]
X_train = train_data.drop("label", axis=1)
X_test = test_data.drop("label", axis=1)
```

**Listing 5.2: Example ML pipeline - data retrieval**

5. After retrieving the required data, she needs to preprocess them, ensuring that missing values are filled or removed, categorical features are encoded etc (Listing 5.3).

```
impute_and_one_hot_encode = Pipeline(
    [
        ("impute", SimpleImputer(strategy="most_frequent")),
        ("encode", OneHotEncoder(sparse=False, handle_unknown="ignore")),
    ]
)
featurisation = ColumnTransformer(
    transformers=[
        (
            "impute_and_one_hot_encode",
            impute_and_one_hot_encode,
            ["smoker", "county", "race"],
        ),
        ("numeric", StandardScaler(), ["num_children", "income"]),
    ],
    remainder="drop",
)
```

**Listing 5.3: Example ML pipeline - data preprocessing**

6. Inspection `Explainer` needs as input a test sample of the dataset, the feature names, a list of what explainability methods should be taken into account in runtime execution of the pipeline and also the label distinct values. Dr. Antonia has access to all this information, so she executes the MLInspect `Pipelineinspec-`

tor, setting as inspection the `Explainer`. Dr. Antonia wants to interpret the model using `LIME` and `SHAP` explainability methods (Listing 5.4).

```
inspector_result = (
    PipelineInspector.on_pipeline_from_py_file(EXAMPLE_PIPELINE)
    .add_required_inspection(
        Explainer(
            [ExplainabilityMethodsEnum.LIME, ExplainabilityMethodsEnum.
    SHAP],
            X_t_test.view(np.ndarray),
            featurisation.get_feature_names_out(),
            [False, True],
        )
    )
    .execute()
)
```

**Listing 5.4: Example PipelineInspector initialization with Explainer inspection**

The `Explainer` inspection takes as input the below:

- **Test sample of the dataset**: `X_t_test = featurisation.fit_transform(X_test, y_test)`
- **Feature names**: `featurisation.get_feature_names_out()`
- **Explainability methods**: `ExplainabilityMethodsEnum.LIME`, `ExplainabilityMethodsEnum.SHAP`
- **Label values**: `[False, True]`

7. Using the framework's inspection capabilities, Dr. Antonia finds the DAG node that describes and persists the estimator with its metadata. This node also contains the explainability results (Listing 5.5).

```
inspection_result = dag_node_to_inspection_results[relevant_node][
    Explainer(
        [ExplainabilityMethodsEnum.LIME, ExplainabilityMethodsEnum.SHAP],
        X_t_test.view(np.ndarray),
        featurisation.get_feature_names_out(),
        [False, True],
    )
]
```

**Listing 5.5: Find Explainer results**

8. The results contain the `LIME` as well as the `SHAP` values. Dr. Antonia decides to show these using plots, to better visualise and understand what each method's results are (Listing 5.6).

```
inspection_result[ExplainabilityMethodsEnum.LIME]["results"].
    show_in_notebook()
explainer = inspection_result[ExplainabilityMethodsEnum.SHAP]["explainer"
    ]
```

```
shap_values = inspection_result[ExplainabilityMethodsEnum.SHAP]["results"
    ]
shap.force_plot(
    explainer.expected_value,
    shap_values,
    X_t_test[:2],
    feature_names=featurisation.get_feature_names_out(),
)
shap.summary_plot(
    shap_values,
    X_t_test[:1],
    feature_names=featurisation.get_feature_names_out(),
    plot_type="bar",
)
shap.decision_plot(
    explainer.expected_value,
    shap_values,
    X_t_test[:1],
    feature_names=featurisation.get_feature_names_out(),
    link="logit",
)
```

**Listing 5.6: Visualize shapley values and LIME**

9. Finally, she checks the results and acts accordingly, either by changing the test sample or the explainability methods that she should apply next.

Without even implementing the explainability methods, Dr. Antonia achieved interpreting her model and features. The integration was seamless and very easy to be reproduced, which gave her more time to focus on her model rather than implementing explainability methods all by herself. Dr. Antonia wished to also check other explainability methods. Fortunately, MLInspect allows the referred set of options (Section 4.4), so Dr. Antonia could apply more methods, like `ALE`, `Integrated Gradients` or `PDP`.

**DAG Representation Scenario**. This scenario covers the needs of when the data scientist wants to see how the pipeline can be represented, taking into account the available explainability methods integrated in MLInspect:

1. The data scientist should install MLInspect, executing same command listed in Listing 5.1. She should only have installed prior to executing the tool, `docker` and `make` commands and be inside the source folder of the project. Now, as described in Section 4.4, a container is initialised, containing all the dependencies in a virtual environment, named e.g. `venv`.

2. Then, she gets the jupyter notebook link (e.g. session link) from the previous command's output and pastes it inside a browser.

3. She creates a notebook file, setting also the kernel of the notebook to previously referred `venv`.

4. After initialisation, Dr. Antonia starts by setting up a pipeline, also containing an explainability method referred in Chapter 4. One pipeline could be:

```
"""Predicting which patients are at a higher risk of complications"""

import os
import warnings

import pandas as pd
from scikeras.wrappers import KerasClassifier
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler

from example_pipelines.healthcare.healthcare_utils import
    create_model_predict

from mlinspect.monkeypatching._mlinspect_ndarray import MlinspectNdarray
from mlinspect.utils import get_project_root

# FutureWarning: Sklearn 0.24 made a change that breaks remainder='drop',
# that change will be fixed
# in an upcoming version:
# https://github.com/scikit-learn/scikit-learn/pull/19263
warnings.filterwarnings("ignore")

COUNTIES_OF_INTEREST = ["county2", "county3"]

patients = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "patients.csv",
    ),
    na_values="?",
)
histories = pd.read_csv(
    os.path.join(
        str(get_project_root()),
        "example_pipelines",
        "healthcare",
        "histories.csv",
    ),
    na_values="?",
)

data = patients.merge(histories, on=["ssn"])
complications = data.groupby("age_group").agg(
    mean_complications=("complications", "mean")
)
data = data.merge(complications, on=["age_group"])
```

```
data["label"] = data["complications"] > 1.2 * data["mean_complications"]
data = data[
    [
        "smoker",
        "last_name",
        "county",
        "num_children",
        "race",
        "income",
        "label",
    ]
]
data = data[data["county"].isin(COUNTIES_OF_INTEREST)]
train_data, test_data = train_test_split(data)
y_train = train_data["label"]
y_test = test_data["label"]
X_train = train_data.drop("label", axis=1)
X_test = test_data.drop("label", axis=1)

impute_and_one_hot_encode = Pipeline(
    [
        ("impute", SimpleImputer(strategy="most_frequent")),
        ("encode", OneHotEncoder(sparse=False, handle_unknown="ignore")),
    ]
)
featurisation = ColumnTransformer(
    transformers=[
        (
            "impute_and_one_hot_encode",
            impute_and_one_hot_encode,
            ["smoker", "county", "race"],
        ),
        ("numeric", StandardScaler(), ["num_children", "income"]),
    ],
    remainder="drop",
)

neural_net = KerasClassifier(
    model=create_model_predict,
    epochs=10,
    batch_size=1,
    verbose=0,
    loss="binary_crossentropy",
)
X_t_train: MlinspectNdarray = featurisation.fit_transform(train_data,
    y_train)
X_t_test: MlinspectNdarray = featurisation.fit_transform(X_test, y_test)
neural_net.fit(X_t_train, y_train)
print("Mean accuracy: {}".format(neural_net.score(X_t_test, y_test)))
# Introduce explainability

# LIME
import lime.lime_tabular
```

```
103
104 explainer = lime.lime_tabular.LimeTabularExplainer(
105     X_t_train,
106     mode="classification",
107     feature_names=featurisation.get_feature_names_out(),
108     class_names=[False, True],
109 )
110 result = explainer.explain_instance(X_t_test[0], neural_net.predict_proba
        )
111 result.show_in_notebook()
112
113 # SHAP
114 import shap
115
116 shap.initjs()
117 explainer = shap.KernelExplainer(neural_net.predict, X_t_train)
118 shap_values = explainer.shap_values(X_t_test[:2], nsamples=100)
119 shap.force_plot(
120     explainer.expected_value,
121     shap_values,
122     X_t_test[:2],
123     feature_names=featurisation.get_feature_names_out(),
124 )
125 shap.summary_plot(
126     shap_values,
127     X_t_test[:1],
128     feature_names=featurisation.get_feature_names_out(),
129     plot_type="bar",
130 )
```

**Listing 5.7: Example ML pipeline use case**

5. Then, in order to generate the DAG, she needs to initialise a `PipelineInspector`, patching the explainability methods' packages defined inside the pipeline. As she executes `LIME` and `SHAP` XAI methods, she needs to patch the corresponding python packages using the command `add_custom_monkey_patching_module` (Listing 5.8).

```
1 inspector_result = (
2     PipelineInspector.on_pipeline_from_py_file(EXAMPLE_PIPELINE)
3     .add_custom_monkey_patching_module(patch_lime)
4     .add_custom_monkey_patching_module(patch_shap)
5     .execute()
6 )
```

**Listing 5.8: PipelineInspector setup and execution use case**

6. After executing the inspector, Dr. Antonia can save the DAG representation in an image extension (e.g. `png`). She can thoroughly check the depiction (Figure 5.1) and how each node is linked altogether.

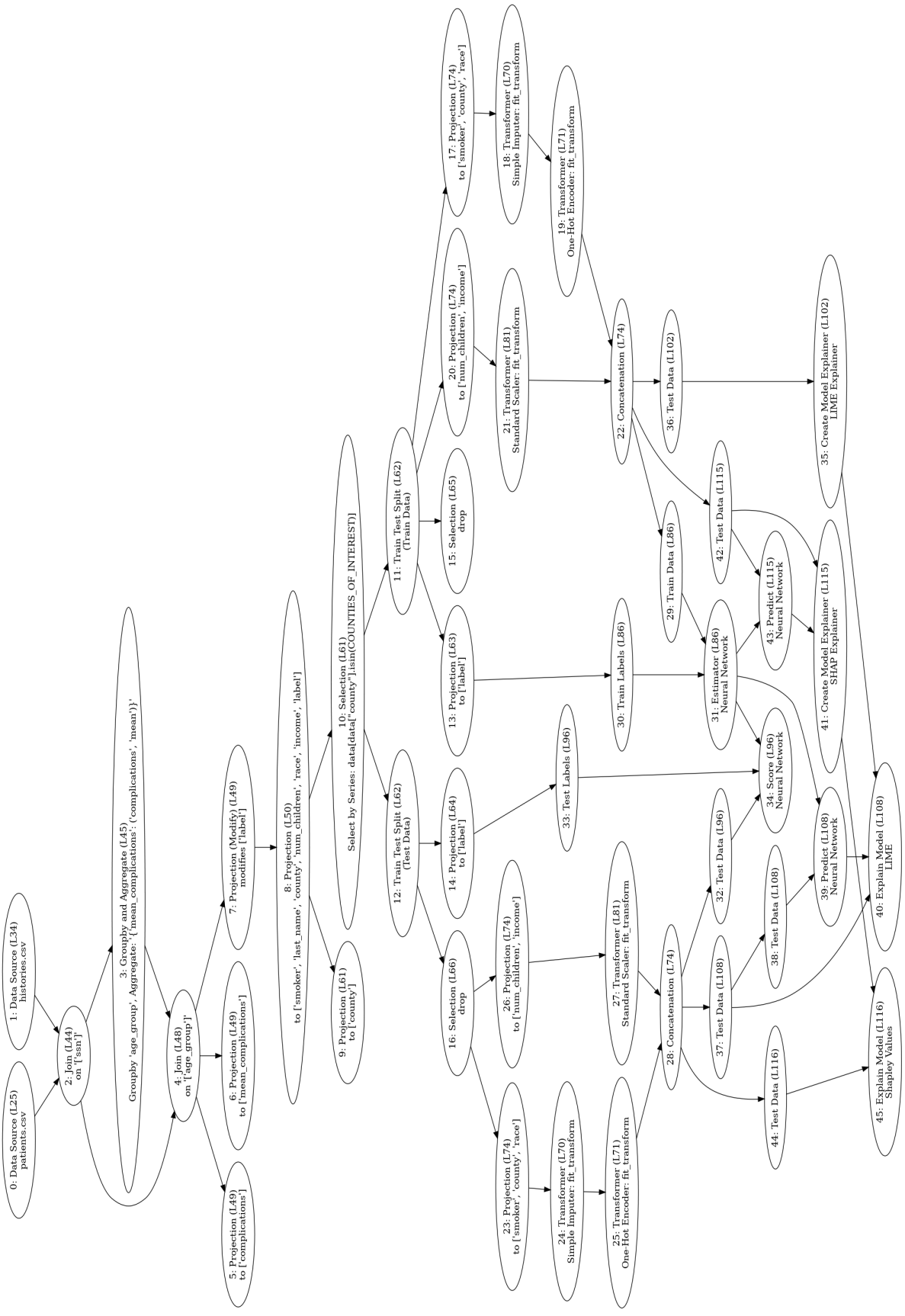



**Figure 5.1: DAG representation scenario, including multiple explainability methods**

7. She understands that `LIME` explainer needs as input test data to be initialised and the predict result of the estimator in order to explain the model. `SHAP` explainer requires the predict result of the estimator as well as test data to be created, but only another set of test data to explain the model.

8. Finally, she can find the relevant node of the explainer output (e.g. 40 or 45 in Figure 5.1) and retrieve the explainer's results accordingly.

This whole scenario helped Dr. Antonia to understand how the pipeline's components are connected together. She, additionally, generated a clear and simple visualisation of the components. Each component does persist metadata or intermediate data, which actually enable her to test the output of each node and additionally check the explainability methods' results with reliability and flexibility.

## 5.1 Unit Tests

Unit tests can be found under the `test/features/explainability` package. For the whole section, the mentioned package will be the root path of all test files.

Regarding the newly introduced inspection, the `/inspections/test_explainer.py` file contains several test cases for the `Explainer` inspection.

1. `test_explainer_healthcare_keras_classifier_all_methods`: This is a test case, which verifies that when subjected to all explainability methods supported and after being trained on healthcare data, a Keras classifier functions properly.

2. `test_explainer_compas_sgd_classifier_all_methods`:This is a test case, which investigates if explainability methods work properly when used on an SGD classifier trained on COMPAS data.

3. `test_explainer_compas_decision_tree_classifier_all_methods`: In this test case, the functionality of all explainability methods is checked when applied to a decision tree classifier trained on COMPAS data.

4. `test_explainer_compas_logistic_regression_all_methods`: This is a typical example of how this method can be utilized to check whether all explainability methods would generate output in logistic regression models trained on COMPAS data or not.

Each of these tests checks the following explainability methods: LIME, SHAP, PDP, ICE, Integrated Gradients, ALE, DALE, and DALEX. Each is supposed to return an explainer and its results applied to machine learning model along with none values indicating no results. It is also verified that the inspection has results for each explainability method, like the explainer and the output of each explainer.

The `/backends/test_explainability_backend.py` file contains several test cases for the explainability backend. In each test case, the functionality of the explainability backend is checked in the context of different types of annotations and multiple analyzers. The tests ensure that the backend correctly propagates the annotations and works correctly with multiple analysers. Specifically:

1. `test_explainability_backend_random_annotation_propagation`: This is a test case that checks if random annotation strategy can be successfully applied in the explainability back-end. It reads the code from the `EXPLAINABILITY_LIME_PY` file and runs the `run_random_annotation_testing_analyzer` function on it. The test asserts that the length of the result is 25, indicating that 25 random annotations were successfully propagated.

2. `test_explainability_backend_row_index_annotation_propagation`: It is a check to see whether row index annotation propagation is correctly handled by the explainability backend. It reads the code from the `EXPLAINABILITY_LIME_PY` file and runs the `run_row_index_annotation_testing_analyzer` function on it. The test asserts that the length of the result is 25, indicating that 25 row index annotations were successfully propagated.

3. `test_explainability_backend_annotation_propagation_multiple_analyzers`: A feature of multiple analyzers is tested in this case when several are used at the same time. It reads the code from the `EXPLAINABILITY_LIME_PY` file and runs the `run_multiple_test_analyzers` function on it. The test then asserts that each analyzer is present in the inspection results for each node in the DAG (Directed Acyclic Graph).

The `/monkey_patching/` package is the most important one that tests if the monkey patching of extra libraries, used for explainability, can be achieved and interpreted in a DAG. This package contains six modules, each testing a different library. For example, the `/monkey_patching/test_patch_alibi.py` verifies that a DAG representation can be achieved, if a code contains `alibi` specific functions used for initialising an explainer and outputting explainability results. The assertion on DAG representation is a way to ensure that some graph matches the pipeline execution graph. In provided code of referred module, DAG is constructed with nodes representing different stages of pipeline such as classifier, creation of explainer, test data explainability, and explainability results. Edges are added so as to join these nodes in order of execution. For example, the classifier is connected to the explainer creation, and the explainer creation is connected to the explainability. The test data explainability is also connected to the explainability node.

After constructing the expected DAG, it is compared with the actual DAG obtained from the pipeline execution. This is done using the `compare` function from the `testfixtures` library. If the actual DAG and the expected DAG are not the same, the `compare` function will raise an exception, causing the test to fail.

Here's a brief explanation of the DAG nodes and edges in the provided code:

- `expected_classifier`: This node represents the classifier used in the pipeline. In the provided tests, different types of classifiers are used, such as `KerasClassifier`, `SGDClassifier`, `DecisionTreeClassifier`, and `LogisticRegression`.
- `expected_explainer_creation`: This node represents the creation of the explainer. In the provided tests, `ALE` and `IntegratedGradients` explainers from the `alibi` library are used.
- `expected_test_data_explainability`: This node represents the test data used for explainability.
- `expected_explainability`: This node represents the explainability process.

The package also includes tests for `lime`, `shap`, `dale`, `dalex` and `sklearn_inspection` libraries, applying the same logic to check that monkey patching and DAG representation works as referenced in Chapter 4.

# 6. CONCLUSIONS AND FURTHER WORK

At the outset of this research, the aim was to help users of MLInspect interpret and understand better their models. This was achieved by integrating explainability methods in MLInspect, making it an ”Explainable MLInspect”. The methods were integrated seamlessly into two aspects of the framework, the newly created inspection `Explainer` and the capability to generate a DAG of a user pipeline that depicts also the explainability methods.

The implementation revealed that it is very important to utilise explainability in user pipelines, to better comprehend how the model functions. This was transparently done by persisting explainability data at the runtime execution of the pipeline, thoroughly decreasing the time to manually do implement this in a pipeline. It can as well narrow down the code of a pipeline as the explainability methods are executed at the background, at runtime. The framework can explain a model with multiple methods as seen in Section 4.2 at the same time, maintaining the data in an easily refined manner to help the user find the explainability results and depict them accordingly, using plots etc. The user can define and monkey patch more explainability methods to be executed at the background as referred in Section 4.4. Another great win was the illustration of a user pipeline and how the components interact with each other, having thus a better picture of what a pipeline does, showing explainability components too.

While this paper significantly integrates multiple packages that support a great aspect of explainability methods, instead it could be further enhanced with more methods or kind of explainers. There exist multiple explainability methods (as enumerated in Section 2.3.2) and in this study it was decided to integrate the most known ones with their own package. More tools (defined in Section 2.3.4) can be also taken into account to have a more comprehensive and complete implementation. Another improvement that can be thought about as a future work is a way to better setup the inspection inputs. These arguments can be further refined to be user defined and more in preference. Finally, this whole interface could be deployed in an actual online website that executes a user uploaded pipeline with the MLInspect framework, where the user has the opportunity to opt for a set of explainability methods to be integrated in the pipeline.

In conclusion, this paper highlights the usage and meaning of explainability methods in a user pipeline. By integrating explainability in MLInspect framework, the user can easily understand how the model works, explain it at runtime, check the pipeline’s components in a DAG and incorporate more explainability methods at their stead. This could save a meaningful time of model interpretability maintenance that is essential in the everyday work of a data scientist.

# 7. FIGURES AND TABLES

# ABBREVIATIONS - ACRONYMS

| | |
|---|---|
| ALE | Accumulated Local Effect |
| AST | Abstract Syntax Tree |
| CD | Continuous Deployment |
| CEM | Contrastive Explanations Method |
| CI | Continuous Integration |
| CT | Continuous Training |
| DALE | Differential Accumulated Local Effects |
| DAG | Directed Acyclic Graph |
| DeepLIFT | Deep Learning Important Features |
| DevOps | Development Operations |
| EBM | Explainable Boosting Machine |
| GDPR | General Data Protection Regulation |
| GIRP | Global Interpretation via Recursive Partitioning |
| ICE | Individual Conditional eXpectation |
| IG | Integrated Gradients |
| LIME | Local Interpretable Model-Agnostic Explanations |
| LOCO | Leave One Column Out |
| LRP | Layer-wise relevance propagation |
| ML | Machine Learning |
| MLOps | Machine Learning Operations |
| PDP | Partial Dependence Plots |
| SHAP | SHapley Additive exPlanations |
| UAT | User Acceptance Testing |
| XAI | EXplainable Artificial Intelligence |

# APPENDIX A. FIRST APPENDIX

# APPENDIX B. SECOND APPENDIX

# REFERENCES


[1] S. Grafberger, P. Groth, J. Stoyanovich, and S. Schelter, “Data distribution debugging in machine learning pipelines,” *The VLDB Journal*, vol. 31, no. 5, pp. 1103–1126, Sep. 2022. [Online]. Available: https://link.springer.com/10.1007/s00778-021-00726-w

[2] A. Kontaxakis, D. Sacharidis, A. Simitsis, A. Abell, and S. Nadal, “HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning,” *ICDE*, 2024.

[3] R. Jabbari, N. Bin Ali, K. Petersen, and B. Tanveer, “What is DevOps?: A Systematic Mapping Study on Definitions and Practices,” in *Proceedings of the Scientific Workshop Proceedings of XP2016*. Edinburgh Scotland UK: ACM, May 2016, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/2962695.2962707

[4] M. Senapathi, J. Buchan, and H. Osman, “Devops capabilities, practices, and challenges: Insights from a case study,” *CoRR*, vol. abs/1907.10201, 2019. [Online]. Available: http://arxiv.org/abs/1907.10201

[5] “ml-ops.org.” [Online]. Available: https://ml-ops.org/

[6] MLOps: Continuous delivery and automation pipelines in machine learning | cloud architecture center. [Online]. Available: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

[7] R. Kuprieiev, skshetry, P. R. (□□□), D. Petrov, P. Redzyński, C. da Costa-Luis, D. de la Iglesia Castro, A. Schepanovski, I. Shcheklein, D. Berenbaum, Gao, B. Taskaya, J. Orpinel, F. Santos, Daniele, R. Lamy, A. Sharma, Z. Kaimuldenov, D. Hodovic, N. Kodenko, A. Grigorev, Earl, N. Dash, G. Vyshnya, maykulkarni, M. Hora, Vera, and S. Mangal, “Dvc: Data version control - git for data & models,” Jun. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11445998

[8] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, “Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics,” 2021.

[9] Neptune team, “neptune.ai,” Feb. 2019. [Online]. Available: https://neptune.ai/

[10] C. Yang, S. Sheng, A. Pham, S. Zhao, S. Lee, B. Jiang, F. Dong, X. Guan, and F. Ming, “BentoML: The framework for building reliable, scalable and cost-efficient AI application.” [Online]. Available: https://github.com/bentoml/bentoml

[11] J. VanderPlas, B. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert, “Altair: Interactive statistical visualizations for python,” *Journal of Open Source Software*, vol. 3, no. 32, p. 1057, 2018. [Online]. Available: https://doi.org/10.21105/joss.01057

[12] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, “Vega-lite: A grammar of interactive graphics,” *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 341–350, 2017.

[13] A. Saucedo, “Awesome Production Machine Learning,” Dec. 2022, original-date: 2018-08-15T14:28:41Z. [Online]. Available: https://github.com/EthicalML/awesome-production-machine-learning

[14] R. Dwivedi, D. Dave, H. Naik, S. Singhal, R. Omer, P. Patel, B. Qian, Z. Wen, T. Shah, G. Morgan, and R. Ranjan, “Explainable AI (XAI): Core Ideas, Techniques, and Solutions,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 194:1–194:33, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3561048

[15] D. Gopinath, “Picking an explainability technique,” Oct. 2021. [Online]. Available: https://towardsdatascience.com/picking-an-explainability-technique-48e807d687b9

[16] E. Onose, “Explainability and Auditability in ML: Definitions, Techniques, and Tools,” Jul. 2022. [Online]. Available: https://neptune.ai/blog/explainability-auditability-ml-definitions-techniques-tools

[17] T. Speith, “A Review of Taxonomies of Explainable Artificial Intelligence (XAI) Methods,” in *2022 ACM Conference on Fairness, Accountability, and Transparency*. Seoul Republic of Korea: ACM, Jun. 2022, pp. 2239–2250. [Online]. Available: https://dl.acm.org/doi/10.1145/3531146.3534639

[18] G. Tripepi, K. Jager, F. Dekker, and C. Zoccali, “Linear and logistic regression analysis,” *Kidney International*, vol. 73, no. 7, pp. 806–810, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0085253815530895

[19] M. T. Ribeiro, S. Singh, and C. Guestrin, “Anchors: High-precision model-agnostic explanations,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/11491

[20] A. Dhurandhar, P.-Y. Chen, R. Luss, C.-C. Tu, P. Ting, K. Shanmugam, and P. Das, “Explanations based on the missing: Towards contrastive explanations with pertinent negatives,” in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2018/file/c5ff2543b53f4cc0ad3819a36752467b-Paper.pdf

[21] S. Wachter, B. Mittelstadt, and C. Russell, “Counterfactual explanations without opening the black box: Automated decisions and the gdpr,” 2018. [Online]. Available: https://arxiv.org/abs/1711.00399

[22] A.-H. Karimi, G. Barthe, B. Schölkopf, and I. Valera, “A survey of algorithmic recourse: definitions, formulations, solutions, and prospects,” 2021. [Online]. Available: https://arxiv.org/abs/2010.04050

[23] S. Verma, V. Boonsanong, M. Hoang, K. E. Hines, J. P. Dickerson, and C. Shah, “Counterfactual explanations and algorithmic recourses for machine learning: A review,” 2022. [Online]. Available: https://arxiv.org/abs/2010.10596

[24] A. Shrikumar, P. Greenside, and A. Kundaje, “Learning important features through propagating activation differences,” 2019. [Online]. Available: https://arxiv.org/abs/1704.02685

[25] H. Nori, S. Jenkins, P. Koch, and R. Caruana, “InterpretML: A Unified Framework for Machine Learning Interpretability,” Sep. 2019, arXiv:1909.09223 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1909.09223

[26] C. Yang, A. Rangarajan, and S. Ranka, “Global model interpretation via recursive partitioning,” 2018. [Online]. Available: https://arxiv.org/abs/1802.04253

[27] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, and K.-R. Müller, *Layer-Wise Relevance Propagation: An Overview*. Cham: Springer International Publishing, 2019, pp. 193–209. [Online]. Available: https://doi.org/10.1007/978-3-030-28954-6_10

[28] A. R. R. J. T. Jing Lei, Max G’Sell and L. Wasserman, “Distribution-free predictive inference for regression,” *Journal of the American Statistical Association*, vol. 113, no. 523, pp. 1094–1111, 2018. [Online]. Available: https://doi.org/10.1080/01621459.2017.1307116

[29] M. D. Morris, “Factorial sampling plans for preliminary computational experiments,” *Technometrics*, vol. 33, no. 2, pp. 161–174, 1991. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00401706.1991.10484804

[30] A. Dhurandhar, K. Shanmugam, R. Luss, and P. Olsen, “Improving simple models with confidence profiles,” 2018. [Online]. Available: https://arxiv.org/abs/1807.07506

[31] K. S. Gurumoorthy, A. Dhurandhar, G. Cecchi, and C. Aggarwal, “Efficient data representation by selecting prototypes with importance weights,” 2019. [Online]. Available: https://arxiv.org/abs/1707.01212

[32] H. Yang, C. Rudin, and M. Seltzer, “Scalable bayesian rule lists,” 2017. [Online]. Available: https://arxiv.org/abs/1602.08610

[33] M. Craven and J. Shavlik, “Extracting tree-structured representations of trained networks,” in *Advances in Neural Information Processing Systems*, D. Touretzky, M. Mozer, and M. Hasselmo, Eds., vol. 8. MIT Press, 1995. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1995/file/45f31d16b1058d586fc3be7207b58053-Paper.pdf

[34] C. Molnar, *8.6 Global Surrogate | Interpretable Machine Learning*. [Online]. Available: https://christophm.github.io/interpretable-ml-book/global.html

[35] M. T. Ribeiro, S. Singh, and C. Guestrin, “”why should I trust you?”: Explaining the predictions of any classifier,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, 2016, pp. 1135–1144.

[36] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/8a20a8621978632d76c43dfd28b67767-Paper.pdf

[37] M. Sundararajan, A. Taly, and Q. Yan, “Axiomatic attribution for deep networks,” 2017. [Online]. Available: https://arxiv.org/abs/1703.01365

[38] D. W. Apley and J. Zhu, “Visualizing the effects of predictor variables in black box supervised learning models,” 2019. [Online]. Available: https://arxiv.org/abs/1612.08468

[39] C. Molnar, *8.1 Partial Dependence Plot (PDP) | Interpretable Machine Learning*. [Online]. Available: https://christophm.github.io/interpretable-ml-book/pdp.html#fnref31

[40] A. Goldstein, A. Kapelner, J. Bleich, and E. Pitkin, “Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation,” 2014. [Online]. Available: https://arxiv.org/abs/1309.6392

[41] V. Gkolemis, T. Dalamagas, and C. Diou, “DALE: Differential Accumulated Local Effects for efficient and accurate global explanations,” Oct. 2022, arXiv:2210.04542 [cs]. [Online]. Available: http://arxiv.org/abs/2210.04542

[42] A. Fisher, C. Rudin, and F. Dominici, “All models are wrong, but many are useful: Learning a variable’s importance by studying an entire class of prediction models simultaneously,” 2019. [Online]. Available: https://arxiv.org/abs/1801.01489

[43] M. Staniak and P. Biecek, “Explanations of model predictions with live and breakdown packages,” *The R Journal*, vol. 10, no. 2, p. 395, 2019. [Online]. Available: http://dx.doi.org/10.32614/RJ-2018-072

[44] J. Klaise, A. Van Looveren, G. Vacanti, and A. Coca, “Alibi Explain: Algorithms for Explaining Machine Learning Models,” *Journal of Machine Learning Research*, vol. 22, no. 181, pp. 1–7, Jun. 2021. [Online]. Available: http://jmlr.org/papers/v22/21-0017.html

[45] V. Arya, R. K. E. Bellamy, P.-Y. Chen, A. Dhurandhar, M. Hind, S. C. Hoffman, S. Houde, Q. V. Liao, R. Luss, A. MojsiloviÄ‡, S. Mourad, P. Pedemonte, R. Raghavendra, J. T. Richards, P. Sattigeri, K. Shanmugam, M. Singh, K. R. Varshney, D. Wei, and Y. Zhang, “Ai explainability 360: An extensible toolkit for understanding data and machine learning models,” *Journal of Machine Learning Research*, vol. 21, no. 130, pp. 1–6, 2020. [Online]. Available: http://jmlr.org/papers/v21/19-1035.html

[46] V. Arya, R. K. E. Bellamy, P.-Y. Chen, A. Dhurandhar, M. Hind, S. C. Hoffman, S. Houde, Q. V. Liao, R. Luss, A. Mojsilović, S. Mourad, P. Pedemonte, R. Raghavendra, J. Richards, P. Sattigeri, K. Shanmugam, M. Singh, K. R. Varshney, D. Wei, and Y. Zhang, “One explanation does not fit all: A toolkit and taxonomy of ai explainability techniques,” 2019. [Online]. Available: https://arxiv.org/abs/1909.03012

[47] J. Wexler, M. Pushkarna, T. Bolukbasi, M. Wattenberg, F. Viégas, and J. Wilson, “The what-if tool: Interactive probing of machine learning models,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 56–65, 2020.

[48] P. Biecek, “Dalex: Explainers for complex predictive models in r,” *Journal of Machine Learning Research*, vol. 19, 11 2018.

[49] H. Baniecki, W. Kretowicz, P. Piatyszek, J. Wisniewski, and P. Biecek, “dalex: Responsible machine learning with interactive explainability and fairness in python,” *Journal of Machine Learning Research*, vol. 22, no. 214, pp. 1–7, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1473.html

[50] S. M. Lundberg, B. Nair, M. S. Vavilala, M. Horibe, M. J. Eisses, T. Adams, D. E. Liston, D. K.-W. Low, S.-F. Newman, J. Kim *et al.*, “Explainable machine-learning predictions for the prevention of hypoxaemia during surgery,” *Nature Biomedical Engineering*, vol. 2, no. 10, p. 749, 2018.

[51] B. Bashari Rad, H. Bhatti, and M. Ahmadi, “An introduction to docker and analysis of its performance,” *IJCSNS International Journal of Computer Science and Network Security*, vol. 173, p. 8, 03 2017.

[52] A. Bhattacharyea, “Creating a Python Makefile,” Jul. 2021. [Online]. Available: https://earthly.dev/blog/python-makefile/

[53] T. Kluyver, B. Ragan-Kelley, P&#233, F. Rez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, n, S. Abdalla, C. Willing, and J. D. Team, “Jupyter Notebooks – a publishing format for reproducible computational workflows,” in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 2016, pp. 87–90. [Online]. Available: https://ebooks.iospress.nl/doi/10.3233/978-1-61499-649-1-87

[54] “pre-commit.” [Online]. Available: https://pre-commit.com/index.html

[55] S. Eustace and The Poetry contributors, “Poetry: Python packaging and dependency management made easy.” [Online]. Available: https://github.com/python-poetry/poetry

[56] “Monkeying Around with Python: A Guide to Monkey Patching,” Feb. 2024. [Online]. Available: https://dev.to/karishmashukla/monkeying-around-with-python-a-guide-to-monkey-patching-obc