Master Thesis

BDMA – Big Data Management and Analytics

On The Problem of Software Quality In Machine Learning Systems

Haftamu Hailu Tefera

Matriculation No.: 0452835

Supervisors :	Prof. Dr. Volker Markl
	Prof. Dr. Odej Kao
Advisor :	Juan Soto
Examiners :	Prof. Esteban Zimányi
	Prof. Oscar Romero

November 12, 2022

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

November 2022, Berlin

. . . . . .

Haftamu Hailu Tefera

Zusammenfassung

Die Verbreitung des maschinellen Lernens (ML) in sicherheitskritischen Bereichen hat die Entwicklung angemessener und geeigneter Testverfahren zur Bewertung ihrer Qualität und Zuverlässigkeit vorangetrieben. Trotz der Fortschritte bei der Softwareprüfung gibt es noch viel zu tun, um ML-Systeme zu testen. Dies liegt an den unterschiedlichen Fehlertypen, wie z. B. Fehler in den Daten, Lernprogrammen oder ML-Frameworks.

In dieser Arbeit wird eine systematische Übersicht über bestehende Forschungsarbeiten im Bereich des Softwaretests für maschinelle Lernsysteme gegeben. Wir untersuchen häufig auftretende Fehler mit Hilfe von statischen Code-Analyse-Tools in vier öffentlich verfügbaren ML-Frameworks (d.h. Scikit-learn, Tensorflow, Keras und PyTorch). Wir führen eine eingehende Analyse bestehender Bugs in Open-Source-ML-Systemen wie Tensorflow, Keras, Scikit-learn, Theano, Spark MLlib, Torch, Mahout und H2O durch, indem wir frühere Stack Overflow-Beiträge von ML- und KI-Experten verwenden. Darüber hinaus untersuchen wir drei neuartige Testverfahren für Systeme des maschinellen Lernens: Rauchtests für Klassifizierungsaufgaben, metamorphe Tests für Bildklassifizierungsprobleme und Verhaltenstests für NLP-Modelle zur Stimmungsanalyse.

In dem Bemühen, bisher unentdeckte Fehler zu entdecken, mögliche Fehler zu minimieren und Diskrepanzen zu finden, haben wir eine Machine-Learning-Anwendung entworfen und implementiert, um siebzehn Klassifizierungsalgorithmen aus Scikit-learn, PySpark und Keras zu untersuchen und zu bewerten. Wir setzten eine systematische und einheitliche Datenvorverarbeitung und Modell-Pre-Train-Tests ein, um die Qualität der Daten sicherzustellen. Zusätzlich implementierten wir Modell-Post-Train-Testfälle unter Verwendung von invarianten Tests und direktionalen Erwartungstests, um zu untersuchen, wie das Modell auf Änderungen bei relevanten und irrelevanten Merkmalen reagiert, während die anderen Merkmale konstant gehalten werden. Bei den empirischen Experimenten wurden keine beobachtbaren Unterschiede festgestellt, die auf einen bisher unentdeckten Fehler hindeuten. Die Scikit-learn-Klassifikationsmodelle übertreffen die entsprechenden Pyspark-Modelle geringfügig, jedoch sind die Leistungsunterschiede zwischen den Modellen der verschiedenen Frameworks vernachlässigbar.

Abstract

The prevalence of machine learning (ML) in safety-critical domains has driven the development of appropriate and suitable testing techniques to evaluate their quality and reliability. Despite advancements in software testing, there is still a lot of work to be done for testing ML systems. This is due to the varying bug types, such as errors in the data, learning programs, or ML frameworks.

In this thesis, we systematically review existing research works in the software testing domain for machine learning systems. We examine frequently occurring bugs with the help of static code analysis tools across four publicly available ML frameworks (i.e. Scikit-learn, Tensorflow, Keras, and PyTorch). We conduct an in-depth analysis of existing bugs across open-source ML systems, such as Tensorflow, Keras, Scikit-learn, Theano, Spark MLlib, Torch, Mahout, and H2O using Stack Overflow past posts from ML and artificial intelligence (AI) practitioners. Furthermore, we examine three novel machine learning system testing techniques: smoke testing applied to classification tasks, metamorphic testing on image classification problems, and model behavioral testing for the NLP sentiment analysis models: RoBERTa pretrained model, Google NLP model, Amazon comprehend, and Microsoft text analytics.

In an effort to discover previously undisclosed bugs, minimize possible bugs, and find discrepancies, we designed and implemented a machine-learning application to examine and evaluate seventeen classification algorithms from Scikit-learn, PySpark, and Keras. We employed systematic and uniform data preprocessing and model pre-train testing to assert the quality of data. In addition, we implemented model post-train test cases using invariant testing and directional expectation tests to examine how the model reacts to changes in relevant and irrelevant features while keeping the other features constant. From the empirical experiments, no observable differences were found to provide evidence of a previously undiscovered bug. The scikit-learn classification models slightly outperform the corresponding pyspark models, however, the variation in performance of the models from varying frameworks is negligible.

Acknowledgments

Completing this thesis would not have been possible without the tremendous support of several people.

First and foremost, I would like to express my gratitude to my advisor, Juan Soto (MSc.) who supported me during the writing of this thesis from the very beginning with helpful advice, suggestions, and constructive criticism. You were an outstanding mentor to me, and I appreciate the motivation, effort, and sympathy you brought. I benefited a lot from working with you and having your scientific and practical advice available to me throughout my thesis work. Secondly, my sincere appreciation extends to the BDMA coordinator, Professor Esteban Zimanyi, and other members of the BDMA committee for their commitment to achieving this goal. Thirdly, thank you to my colleagues at the BDMA program who supported me in writing and encouraged me to achieve my goal. I am grateful for their support. Finally, my very special gratitude extends to my family for their constant encouragement, love, and support. Thank you!

Contents

List of Figures

List of Tables

List of Abbreviations

ML ..... Machine Learning

AI ..... Artificial Intelligence

DL ..... Deep Learning

SVMs ..... Support Vector Machines

MT ..... Metamorphic Testing

MRs ..... Metamorphic Relations

MR ..... Metamorphic Relation

NLP ..... Natural Language Processing

QA ..... Quality Assurance

ANN ..... Artificial Neural Network

1 Introduction

The term Artificial Intelligence(AI) is the broad science of mimicking human abilities. On the other hand, Machine Learning(ML) is a specific subset of modern AI that trains a machine how to learn from data. ML is one method of data analysis that automates the process of a model building using algorithms to improve performance using data. ML systems are generally successful from a performance perspective, however, their quality and reliability are still a major challenge. The lack of tools and established best practices and standards for testing ML systems results in many ML models not being used as expected by the end-user [1, 44, 92, 73]. ML systems are different from traditional systems in that they are stochastic and data-dependent decision logic. This poses new challenges, in terms of reliability, testing, and ensuring quality.

1.1 Motivation

The prevalent adoption of ML in safety-critical systems like autonomous driving and healthcare has driven the development of testing methods to evaluate and assess their quality [92, 73]. These novel discoveries pave the way for a new line of research, machine learning testing, and bug detection for learning programs. Despite advancements in software testing, there is still a lot of work to be done for testing ML systems. This is due to the fact that in ML systems, both the source code and the data dictate the behavior of the ML system. Bugs(faults) in ML systems can be found in the data, in the learning programs, and in the frameworks. Due to this, standard software testing techniques are insufficient and inefficient for testing such systems [92, 73].

Performance-based metrics were used as the primary criterion to evaluate the success and reliability of ML systems. However, these metrics can only reveal a few things about the behavior of ML systems. In addition to that, the use of ML in testing software offers minimal support for testing the learning programs and the training data. Due to this, safety-critical ML systems have raised concerns about the prediction quality[25, 92]. Furthermore, ML testing greatly differs from standard software testing in that it simulates system behavior based on the training data and prediction is stochastic [73, 36]. Thus, ML learning program model

Common Vulnerabilities and Exposures(CVE)
CVE is a catalog of known security threats to computer systems. Currently, there are more than 141K publicly disclosed security vulnerabilities. CVE serves as a critical security infrastructure across all industries by enabling vulnerabilities to be circulated and referenced with a consistent identifier.
The AI Incident Database (AIID)
AIID is an open-source database designed to help AI researchers and practitioners better understand AI and ML failures. AIID contains around 16K incidents that caused safety, fairness, and other problems collected from the popular press, research, and trade publications
The Risks Digest
The **RISKS Digest** is an online database of software risks and unintended consequences and hazards caused by poor system design. It has been published periodically since 1985 by the Committee on Computers and Public Policy of the Association for Computing Machinery.

Table 1: The **AIID**, **Digest Risks**, and **CVE** Databases [1, 5, 65]

behavioral testing is crucial to uncover training faults, provide users confidence in using ML systems, and choose one ML model over another given the same input data and other model training parameters.

These days, business organizations are increasingly deploying AI systems to safety-critical domains: healthcare, credit scoring, law enforcement, the aviation industry, self-driving cars, education, and corporate recruitment [57]. However, deployed AI systems are prone to unforeseen and often dangerous failures. Failures of such systems pose severe risks to life and expose the limits of intelligent systems used in production. Some of the widely used open-source online databases for documenting and reporting real-world AI, ML, and information systems failures are presented in Table 1.

The AI Incident Database (AIID) aims to improve the safety of intelligent systems across a variety of domains by documenting past incidents. The database contains around 16K incidents, many of them caused by AI and ML algorithms' biases. We looked into facial recognition AI systems' failures, and we found many real-world problems such as a New Zealander of Asian descent in New Zealand who had his passport application rejected after the software that approves photos claimed his eyes were closed, the police department of Michigan wrongfully ar-

rested a black man over false facial recognition match, and Twitter's Photo Crop Algorithm favors white faces and women [1]. Another recent AI-based incident was a Deepfake video of Ukrainian president Volodymyr Zelensky calling on his troops to surrender to Russian forces [1] and a video of the incident made it onto a hacked Ukrainian news site after going viral on Facebook. We examined facial recognition incidents and a few recent harms are shown in Figure 1.

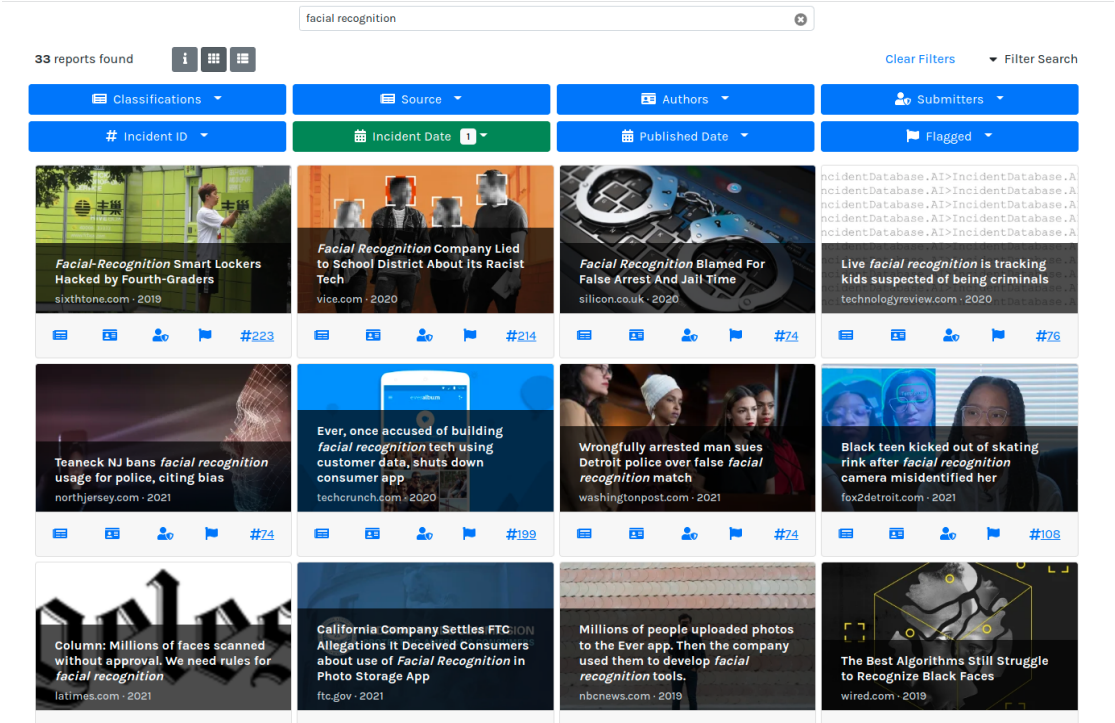

Figure 1: The screenshot shows what happens when a user enters "**facial recognition**" into the "**Discover**" AI Incidence Database search box [1]

From the three databases, we learned that, even though researchers have applied various testing techniques for the training data, the learning program, and other software verification and validation methods to ML systems, there is still a high demand for testing ML systems to ensure quality by detecting bugs early. In addition, open-source ML systems, such as scikit-learn, Keras, PyTorch, and Spark ML are widely used in industry and academia due to their excellent APIs. Prominent research works [25, 73, 56, 36] have reported that a few testing studies have been conducted to detect learning program faults and evaluate their prediction correctness, but no specific confidence studies for supervised ML algorithms have been publicly reported.

After researching the current state-of-the-art solutions [72, 92, 73], we came to know that testing the data, the ML model, and the framework can enable users to identify faults and minimize unforeseen risks. Hence, in this thesis, we want to identify and analyze frequent bugs in models through software code analysis and experimenting with novel testing techniques.

1.2 Problem Statement

The objective of the thesis is to examine, understand, experiment, and answer the following questions to build user trust in using open-source ML systems. First, how users of machine learning systems, including practitioners, researchers, and members of society, can gain confidence in using and deploying machine learning software. Second, how should open-source ML software systems be tested? And how much confidence do users realistically have in publicly-available ML systems to build and deploy ML applications using classification algorithms? Third, what is considered adequate when testing machine learning systems? Fourth, who decides what is appropriate testing for a given ML software and what are the novel ways for testing machine learning, and the unique challenges associated with them? Finally, what makes machine learning systems vastly different from conventional software systems, what are the state-of-the-art solutions for testing machine learning classification models, and how to apply them in practice?

1.3 Objective

We plan to apply concepts of model testing, data and model validation to improve the quality of machine learning systems. We aim to employ model pre-train tests to identify data integrity issues. Furthermore, to ensure that the algorithms are correctly implemented, we write model post-train tests, which evaluate the model's behavior when the data is changed.

The specific subproblems are:

(a) What are representative examples of frequently occurring machine learning bugs?

ML bugs as reported in GitHub fix commit issues, Stack Overflow posts, and software repositories are analyzed to get a sense of frequently-occurring bugs. A codebase analysis of software repositories of open-source frameworks was conducted using static analysis tools. Furthermore, to gain a deeper understanding of bugs, we replicated three works using smoke testing, metamorphic testing, and model behavioral testing.

(b) **What are the ideal data preparation and feature engineering practices for consistent performance?**

Data determine the reliability and quality of the ML system. We performed data integrity checks using explanatory analysis. We performed feature engineering: we input missing values, standardized the data, handle class imbalance, manage duplicate values, encode categorical values, handle outliers, and assemble features into a vector form. We also provide a uniform way of data processing for three libraries.

(c) **What are the latest open-source data and model validation tools? Why is data and model validation so significant in operational ML systems?**

We examine open-source data and model validation tools. In particular, we perform data validation on the train and test data and model validation to evaluate the quality of the data and trained model using a **DeepChecks**

(d) **How can we minimize training an erroneous implementation of a model using pre-training tests?**

In pre-train tests, we want to catch errors in our implementation i.e before we feed the data to the ML model, we perform assertions on the various features of the data to assure the quality of the data and detect bugs that may exist in the data.

(e) **How to evaluate and optimize machine learning models to maximize performance?**

We examine the classification performance metrics of the 17 classifiers with accuracy, precision, recall, and RoC metrics. We work with the metrics to determine which metric is suitable for the problem we want to address. We also perform various hyperparameter tuning and k-fold cross-validation to improve model performance.

(f) **How can we evaluate or ensure the expected behavior of trained machine learning models?**

In model post-training testing, we want to check if the model is behaving correctly i.e if the learned logic works as expected. Thus, we perform an invariant test, and directional expectation tests to check how a model reacts to a change in relevant and irrelevant features keeping other features constant. we also experimented with model evaluation to ensure satisfactory performance.

1.4 Related Work

Testing and monitoring are key considerations for ensuring quality and reducing the technical debt for the production readiness of ML systems. Due to the fact that any model's actual prediction behavior can be difficult to predict in advance, specific tests are difficult to formulate. In a separate study, a group of researchers[25], proposed a set of 28 unique tests and monitoring needs, drawn from experience with a wide range of production ML systems to help quantify these issues and present an easy-to-follow roadmap to ensure quality in production readiness of machine learning systems.

We took inspiration from behavioral testing of NLP models [72] to apply invariant and directional expectation tests to classification models from scikit-learn. Furthermore, the ML Test Score [25], a rubric for ML Production Readiness and Technical Debt Reduction by Google gave us significant insights into what components and features should be tested to improve the overall quality of the machine learning pipeline.

1.5 Thesis Workflow

The thesis workflow is given below.

Figure 2: Thesis Workflow Diagram

2 Foundations

Machine learning (ML) is a topic of study part of artificial intelligence focused on understanding and developing "learning" methods, or methods that use data to improve performance on a set of tasks [60]. The main idea behind ML is learning from examples: we prepare a dataset with examples, and a machine learning system learns from this dataset [39].

Definition 1 (Machine Learning) *Machine learning is defined as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision-making under uncertainty (such as planning how to collect more data!) [63].*

The goal of machine learning is to develop models that get better and learn with experience [60]. In machine learning, algorithms and statistical models are used to analyze and draw inferences from patterns in data to learn and adapt without following explicit instructions [24]. The quality of Machine Learning systems primarily relies on data or models derived from data [82].

2.1 Machine Learning

There are several different approaches to machine learning; these are classified into three major categories depending on the nature of the data: namely, supervised, unsupervised, and reinforcement learning [63, 11].

(a) Supervised Machine Learning

In supervised learning, the goal is to learn a mapping from a set of inputs to outputs, given a labeled set of datasets. We can express a supervised machine learning model mathematically as [39]:

$$\mathrm{y} \approx \mathrm{g}(\mathrm{X})$$

Where,

- g is the function that we want to learn with ML
- X is the feature (typically, a matrix) in which rows are feature vectors
- y is the target variable, a vector

The goal of supervised machine learning is to learn this function g in such a way that when it gets the matrix X, the output is close to the vector y. When we train a model, an algorithm takes in a matrix X in which feature vectors are rows and the desired output is the vector y, with all the values we want to predict. The result of training is g, the model [39].

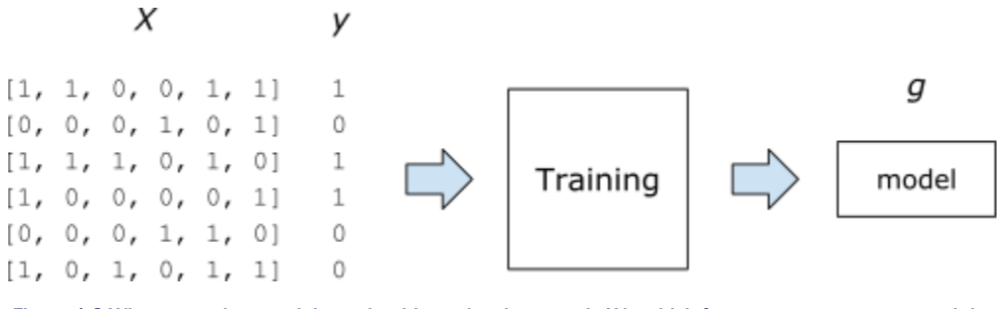

Figure 3: Supervised Machine Learning [39]

Depending on the target variable y, there are different types of supervised learning problems. These are classification, regression, and ranking.

(b) Unsupervised Machine Learning

The second type of machine learning is unsupervised learning or descriptive learning/approach. Here we are only given inputs,

$$D = \{x_i\}_{i=1}^{N}$$

and the goal is to find interesting patterns or relationships in the data. This is often called knowledge discovery. This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use unlike supervised learning, where we can compare our prediction of y with the ground truth in the test set [63]. This machine learning type is very helpful when you need to identify patterns and use data to make decisions.

(c) Reinforcement Learning

In Reinforcement machine learning, an agent receives a reward in the next time step to evaluate its previous action. Deep learning and neural networks

have received a lot of attention recently, mostly because of breakthroughs in computer vision methods.

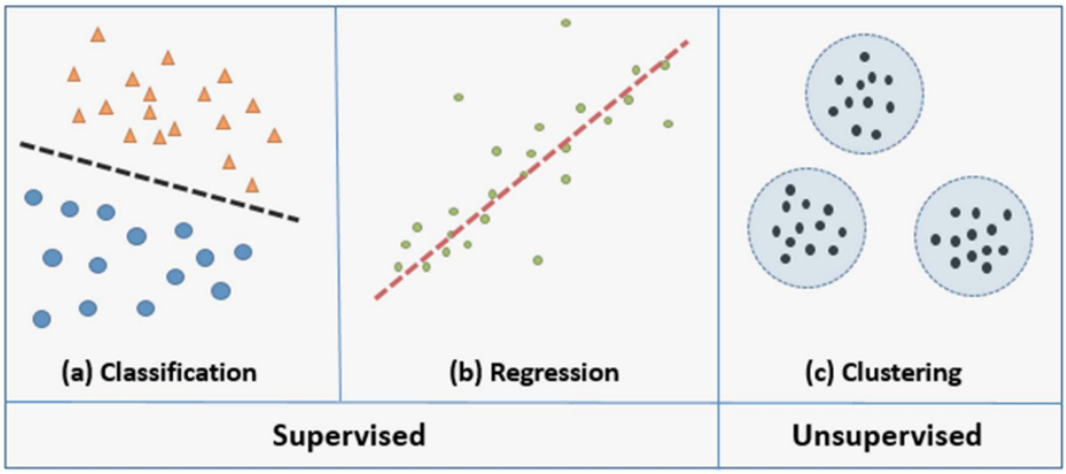

Figure 4: Typical ML tasks. (a) Classification - finding a separating line between the two classes, "circles" and "triangles", (b) Regression - fitting a predictive model, (c) Clustering - grouping a set of samples into a number of clusters [18]

Often the hardest part of solving a machine learning problem can be finding the right algorithm for the problem at hand. The **scikit-learn algorithm cheat sheet** is designed to guide users on how to approach problems with regard to which algorithm to try on the data [3].

2.2 Classification Algorithms

Classification algorithms are used to categorize data into a class. Classification algorithms use features and their relations to each other to define the target variable.

(a) Support Vector Machines

Support Vector Machines(SVMs) are a group of algorithms used for classification and regression. SVMs are effective for high-dimensional data through the use of decision boundaries(hyperplanes) and custom kernels to handle structured and unstructured data. Through careful selection of kernel functions and regularization parameters, SVM can minimize the effect of overfitting. A hyperplane is a separator for dividing the points into their respective classes. A good separation in the classes is achieved by the hyperplane with the largest distance to the nearest training data points. The larger the distance the lower

the generalization error of the model.

Kernel functions are added to a linear SVM classifier to solve non-linear problems.SVM uses kernel functions to transform linearly inseparable data into linearly separable data by creating a line or a hyperplane which separates data into classes [8]. The most common kernel functions used in SVM are Gaussian Kernel, Gaussian Kernel Radial Basis Function (RBF), Sigmoid Kernel, and Polynomial Kernel.

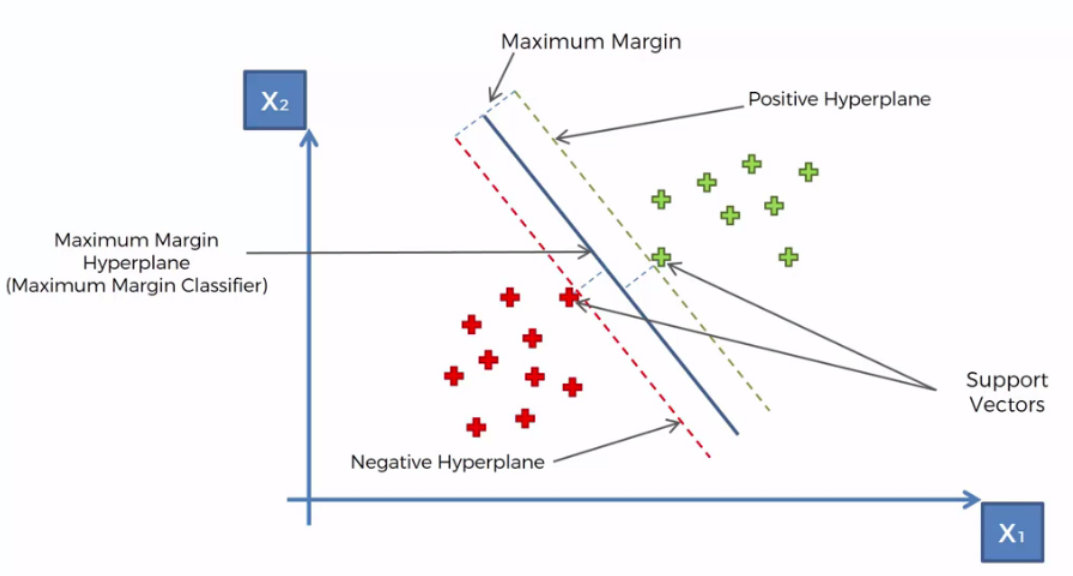

Figure 5: Positive, negative, optimal hyperplanes & maximal margin for SVM [27]

(b) Decision Trees

Decision trees are non-parametric methods used for classification and regression tasks. Decision trees create models that predict target features by learning from the data. The variations of decision trees like CART, random forest, bagged decision trees, and boosted decision trees are among the robust and powerful techniques [26]. The CART decision tree divides the input data and different split points are tried and tested to minimize the cost function and the split with the lowest cost is selected. A greedy approach is used in selecting the input variables and split points to minimize the cost function[26].

(c) Random Forest

Random Forest uses ensemble methods for the predictions of different machine learning algorithms to make more accurate predictions than the predictions

made by each algorithm individually is called bagging [80, 26]. It is a collection of decision trees for the random subsets of the data based on Bagging and bootstrapping methods, and the predictions are based on a majority vote over the predictions of individual trees [80]. The decision trees are constructed using bagging and feature randomness. In bagging, multiple models are combined to produce a single, generalized result. Whereas, Bootstrapping involves creating subsets of observations from the original dataset, with replacement. Classification problems require each tree to vote and the most common class is chosen as the final class.

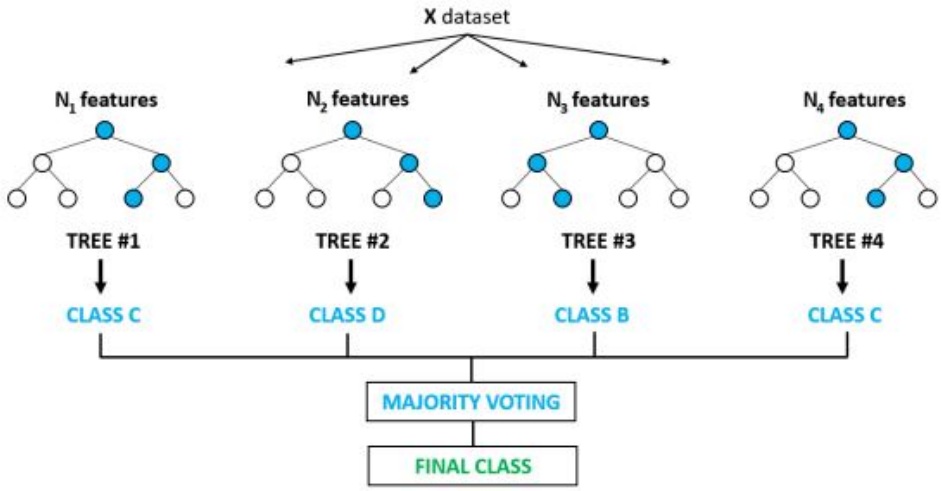

Figure 6: Random Forest algorithm for classification of X dataset [40]

(d) Naive Bayes

Naive Bayes is a statistical classification technique based on Bayes Theorem. The naive Bayes classifier assumes that the effect of a particular feature in a class is independent of other features. Even if the features are interdependent, they are still considered independently. This assumption simplifies computation, and that's why it is considered naive.

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Figure 7: Class conditional independence [64]

Where,

- **P(h)**: the probability of hypothesis h being true (regardless of the data).
- **P(D)**: the probability of the data (regardless of the hypothesis).
- **P(h|D)**: the probability of hypothesis h given the data D.
- **P(D|h)**: the probability of data d given that hypothesis h was true.

(e) Logistic Regression

A logistic regression classifier is a statistical analysis method to predict a binary outcome, such as yes or no, based on prior observations of a data set. The algorithms implement different regularization methods and handle dense and sparse data and calculate the probability of the classes to show the confidence of the predicted value using **sigmoid** function [10]. It is a parametric form for the distribution $P(Y|X)$ where Y is a discrete value and $X = x1, x2, x3. \ldots, xn$ is a vector containing discrete or continuous values [52].

The following equation can be used as a model for logistic regression

$$\hat{Y} = h(X|\{W, b\}) = \sigma(X * W + b)$$

The above model can also be rewritten as

$$P(Y = 1|X) = \frac{1}{1 + e^{-(w_0 + \sum_{n=1}^{n} w_i * X_i)}}$$

$$P(Y = 0|X) = 1 - P(Y = 1|X)$$

The parameter W is selected by maximum-likelihood estimation. A maximum-likelihood estimation estimates how likely it is that Y values will be observed in the training data [52]. The maximum-likelihood estimation is that a search procedure seeks values for the coefficients that minimize error in predicted probabilities with respect to the actual class [52]

$$W \leftarrow \arg\max_{W} \sum_{l} ln\ P(Y^l|X^l, W)$$

The prediction accuracy increases with the removal of noise and highly correlated inputs [52]. Logistic regression assumes a linear relationship between input and output variables [52]. Data transformations to linear relationships can result in more accurate predictions [52].

2.3 Clustering Algorithms

In clustering, data points are divided into groups so they are more similar to other data points in the same group and more dissimilar to those in other groups. It has a large number of applications such as recommendation engines, market segmentation, social network analysis, medical imaging, and anomaly detection. The most widely used clustering algorithms are.

(a) K-Means

K-Means is an unsupervised, prototype-based, partitional clustering technique that aims to partition observations into user-specified K clusters in which each observation belongs to the nearest cluster [58]. The K-means is an approximation to an NP-hard combinatorial optimization problem. The algorithm is iterative in nature and converges, however only a local minimum can be obtained [58]. K-Means calculates the distance between each point and the centroids in order to minimize the cost function. K-means performs poorly when the data contains outliers and when clusters have a variety of sizes, densities, and non-globular shapes [58]. K-Means clustering may cluster loosely related observations together.

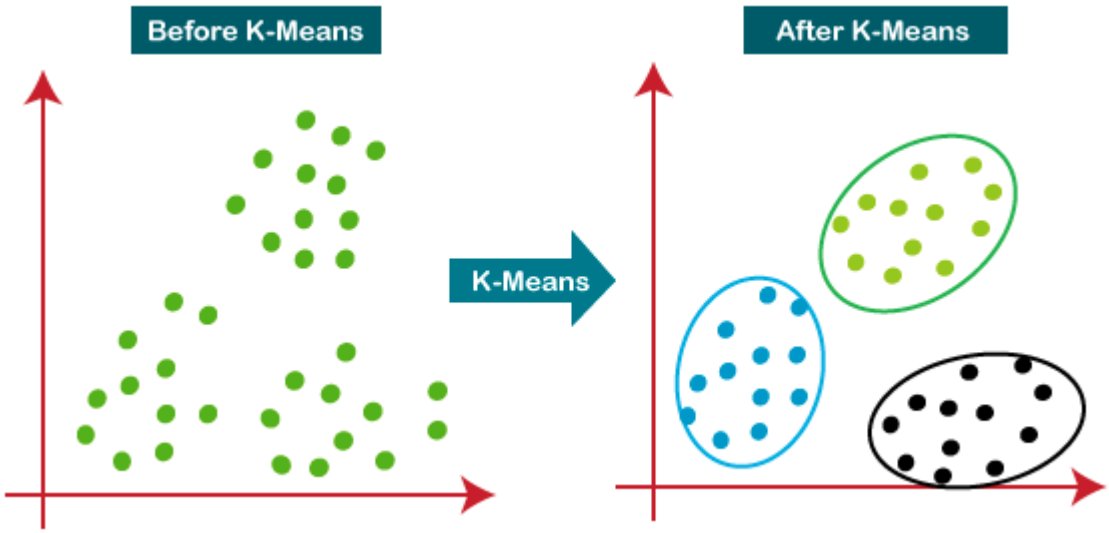

Figure 8: A diagram illustrating how K-means algorithms work [7]

Figure 8 shows three clusters are formed and no points are moving between centroids.

(b) DBSCAN

DBSCAN stands for **D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise, is a density-based clustering algorithm that produces a partitional clustering in which the number of clusters is automatically determined [50]. In low-density regions, points are regarded as noise and are omitted; as a result, DBSCAN does not produce a complete clustering [58].
The DBSCAN algorithm uses two fundamental parameters to cluster data points:

- **minPts**: The minimum number of points (a threshold) clustered together for a region to be considered dense.
- **eps (ϵ)**: A distance measure that will be used to locate the points in the neighborhood of any point

The key idea of DBSCAN is that for each object in a cluster, the neighborhood of a given radius (ϵ) must contain at least a minimum number of objects ($minPts$), which means that the cardinality of the neighborhood must exceed a certain threshold [50].

Algorithm	Advantages and Limitations
Naive Bayes	• Scalable with the number of estimators and data points. • Requires a small amount of training data • Zero probability problem • Assumes that all predictors are independent, which rarely happens
Support Vector Machines	• Scales well to high dimensional data • The kernel trick for complex problems • Have generalization inbuilt • Choosing the best kernel function is not easy • Difficult to interpret the final model, and variable weights
Logistic Regression	• Trained parameters give inference of each feature. • No assumptions about class distributions • Less prone to over-fitting in a low-dimensional data • Efficient for linearly separable datasets • Can't perform well for nonlinear problems • A lack of many observations leads to overfitting
Decision Trees	• Easy to interpret for small-sized trees • Handle continuous and discrete attributes • Relatively easy to understand the prediction process • Less effort for data preparation • Prone to noise and the risk of overfitting is high • Performs poorly if many complex interactions are present • Dataset small variations can lead to a different tree
K-Means	• Fast, robust, and easy to understand • Produce tighter clusters and computationally faster • Gives accurate clusters if the data sets are distinct • Finding the number of clusters is not easy • Not suitable for highly overlapping data • Work only for well-shaped clusters • Does not handle outliers properly
DBSCAN	• The number of clusters need not be specified • Can discover arbitrarily shaped clusters • Find a cluster completely surrounded by other clusters • Robust towards outlier detection • Does not work well in the case of high-dimensional data • Datasets with altering densities can be tricky • Sensitive to clustering parameters minPoints and eps • Not suitable if the dataset is too sparse

Table 2: Summary of classification algorithms

2.4 Machine Learning Lifecycle

Machine learning development is a series of defined steps taken to develop, deploy and monitor a machine learning model. According to CRISP-DM [88], the machine learning process is iterative and it starts with understanding the business, and data understanding, then moves into data preparation, training the model, and evaluating the results, followed by the model deployment. Understanding the trending requirements, we decided to use the architecture in Figure 9, to describe the current ML workflow.

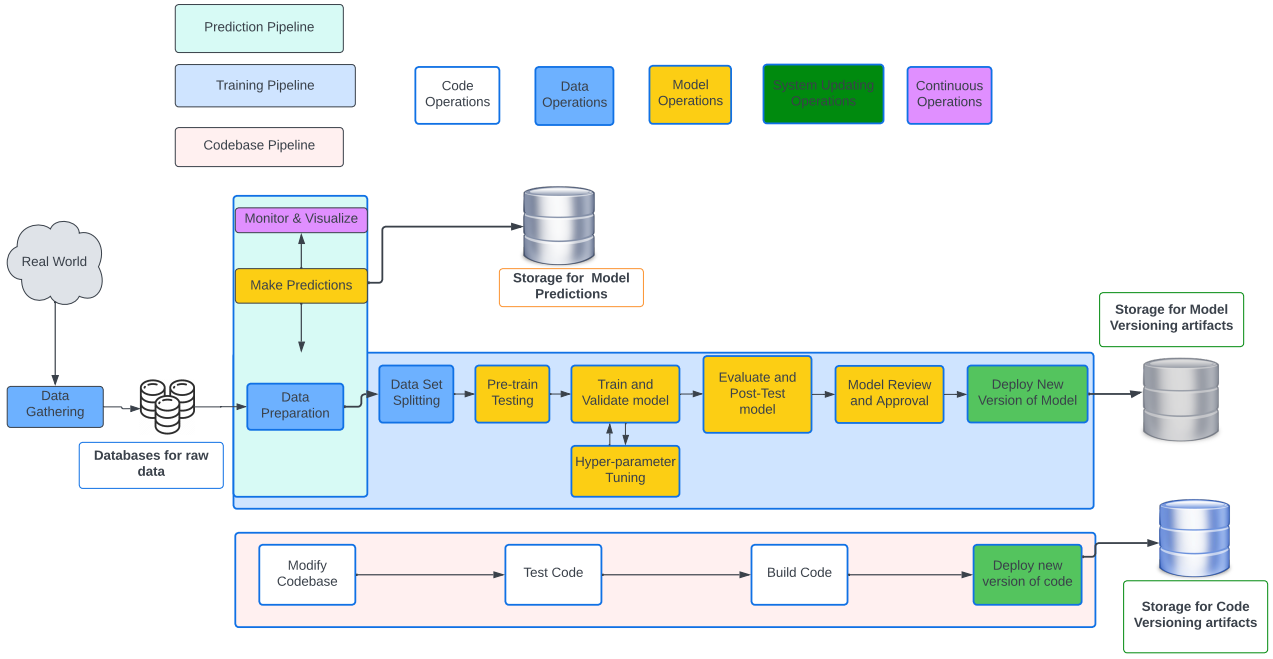

Figure 9: An architecture for a modified machine learning pipeline [51]

The modified architecture consists of three pipelines; codebase pipeline, prediction pipeline, and training pipeline. The prediction pipeline represents the functionality of the system in production. The training pipeline is used to evaluate ML in offline settings. In a highly automated well-designed system the training pipeline can be online and operate with almost no human interaction. The codebase pipeline tracks the code changes using code-tracking tools such as collaborative tools and static analysis tools. The individual components of the architecture are described Figure 9 below.

- **Data Preparation** for collecting and analyzing training data

- **Pre-train tests** used for detecting bugs(errors) before the model is trained
- **Model Training** construct a machine learning model using training data
- **Hyperparameter Tuning** provides different combinations of hyperparameter values to select the best performing models.
- **Model Evaluation** evaluates how well the trained model performs
- **Post-train tests** inspect model behaviors for a variety of relevant scenarios
- **Review and Approval** Investigate evaluation metric scores and ensure correct learned logic and a certain performance level before predictions can be trusted.
- **Prediction** Use the trained model to make predictions on real-world data.
- **Deployment** Puts the model into production environment
- **Monitoring** detect bugs or faults that occur in production and provide insights into when the model should be retrained.

In the following section, we explain the core components of the machine learning life cycle in more detail.

(a) Data Ingestion and Processing

Data ingestion, preprocessing, and validation are the main components that deal with data preparation for ML algorithms. The goal is to clean and transform the data in such a way that it can be used as input to an ML model. This will make sure that the data collected is comprehensive enough to train a reliable model and representative enough of the data that will be encountered in production [61]. In general, this step prepares the data for the succeeding steps in the Pipeline by aggregating data from several sources into a single source of truth. It also cleans data to ensure that all values are in the same format and type. In addition, it imputes missing values, scales data, and extracts features that may be beneficial for the training process.

(b) Model Training

We train a model to take inputs and predict an output with the lowest error possible. Model tuning has seen a great deal of attention lately because it can yield significant performance improvements and provide a competitive edge. A comparison of several models with different settings determines how well a given model can perform for a given problem [51, 61, 39].Model selection is the process

of selecting one final model from among a collection of candidate models. A model selection process can be applied across different types of models as well as between models of the same type but with different hyperparameters

A cross-validation technique is one way to accomplish this by dividing the data set into several parts, each of which is validated separately, while the remaining parts are used for testing. Furthermore, comparisons can also be made with baseline models. Baseline models are simple models that provide reasonable results and do not require much time to build. Linear regression and logistic regression are representative examples of baseline models. A model that does not provide significantly better performance than baseline models may need to be reconsidered.

(c) Model Evaluation

Evaluating the model means measuring how well it performs with respect to a selection of metrics designed to provide insight into the model. The main objective of any predictive modeling technique such as machine learning is to create a model that makes reliable predictions on new data points.

(i) Evaluation of Classification Models

Classifiers can be evaluated in many ways. One of the metrics is called a confusion matrix. In binary classification, there are four outcomes of confusion matrix [15, 66].

- **True Positives (TP)** The model predicted true and it is true.
- **True Negatives (TN)** The model predicted false and it is false.
- **False Positives (FP)** The model predicted True and it is false. A model predicted that someone was sick, but that person was not sick.
- **False Negatives (FN)** The model predicted false and it is true. As an example, the model predicted that someone wasn't sick, but the person was.

From [15, 66], the common metrics for classification models which can be derived from Table 3 includes:

- **Accuracy** is a ratio of correctly predicted observations to the total observations. Accuracy can be a useful metric for balanced data

$$Accuracy = TP + TN \;/\; (TP + TN + FP + FN) \tag{2.1}$$

N=185	Predicted:No	Predicted:Yes	
Actual:No	TN = 50	FP = 20	70
Actual:Yes	FN =15	TP = 100	115
	65	120	

Table 3: A confusion matrix for a binary classifier with N =185

- **Precision And Recall** are used as a measure of relevance when we have an imbalance of data.
  - **Precision** is the fraction of relevant instances among the retrieved instances
  - **Recall** is the fraction of relevant instances that have been retrieved over the total number of instances

$$Precision = TP \,/\, TP + FP \tag{2.2}$$

$$Recall = TP \,/\, TP + FN \tag{2.3}$$

- **F1 Score** is the weighted average of precision and recall. F 1 speaks about the robustness of a given classifier. The greater the F1 value, the better the model [63].

$$F1\ Score = 2 * 1 \,/\, (1/Precision + 1/Recall) \tag{2.4}$$

- **True Positive Rate (TPR)** is the fraction of true positives among all positive examples
- **False Positive Rate (FPR** is the fraction of false positives among all negative examples

(ii) ROC Curve

Receiver operating characteristics or ROC curve is used for visual comparison of classification models which shows the relationship between the true positive rate and the false positive rate. The area under the ROC curve is the measure of the accuracy of the model. ROC curves are great to evaluate imbalanced data. Figure 10 shows how the various classifiers classify the positive class for the titanic dataset using scikit-learn.

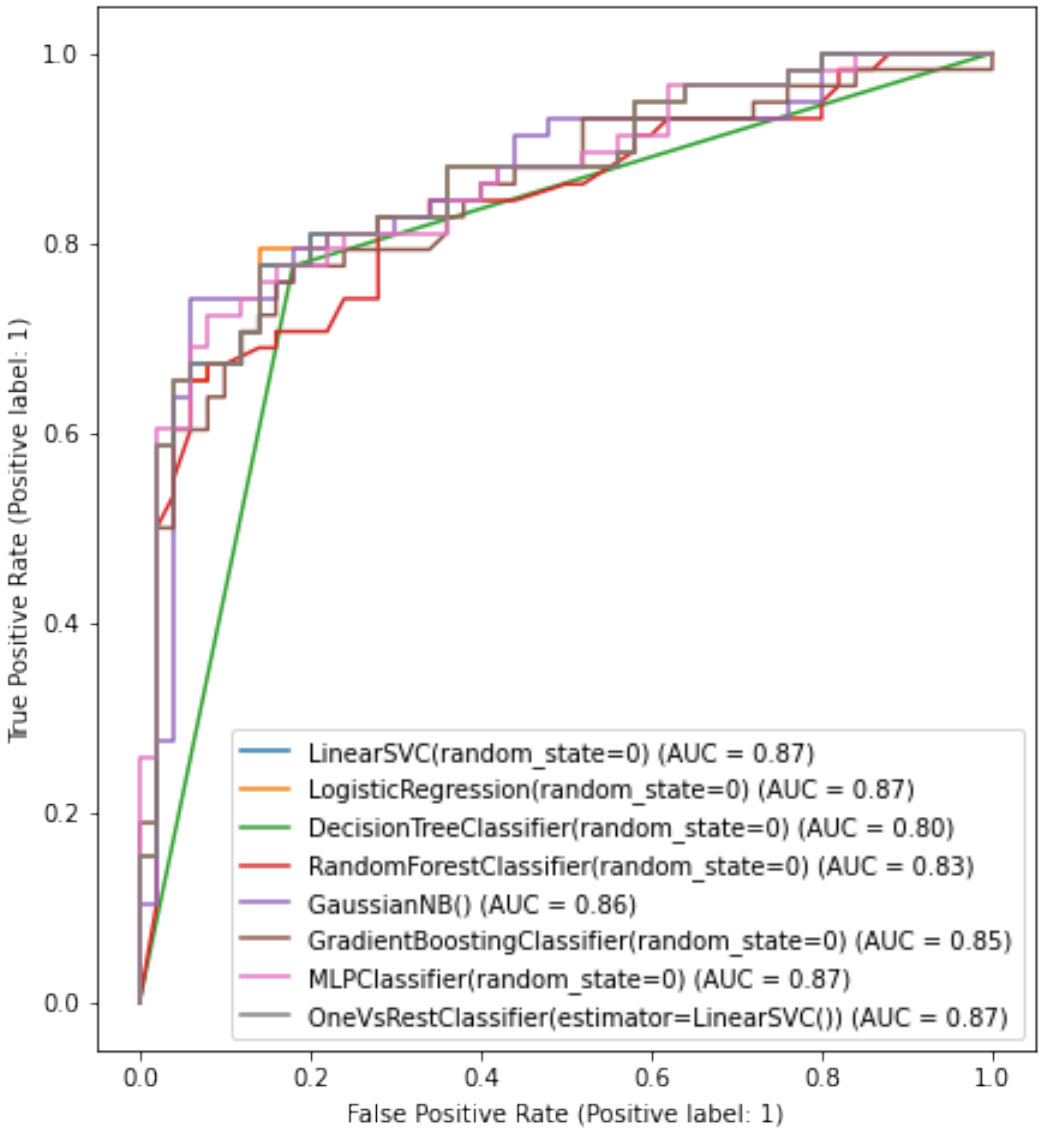

Figure 10: ROC curve for eight classifiers [14]

(iii) Evaluation of Clustering Models

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall as in the case of supervised learning algorithms [15]. Clusters are evaluated based on some similarity or dissimilarity measure such as the distance between cluster points. The Silhouette coefficient is the most popular metric for evaluating clustering algorithms.

Silhouette Coefficient The silhouette Coefficient or silhouette score is a met-

ric used to calculate the goodness of a clustering technique. Its value ranges from -1 to 1. The Silhouette Score is given by

Silhouette Score = (b-a) / max(a,b)
Where ,

- **a** = the average distance between each point within a cluster.
- **b** = the average distance between all clusters.

The silhouette score of -1 corresponds to incorrect clusters and 1 to highly dense clusters. Scores around zero indicate overlapping clusters [15, 23].

Researchers [67, 20], claim that measuring only the accuracy of the model is not sufficient to understand its performance. Instead, performance metrics should reflect audience priorities. Therefore, specific metrics, such as key performance indicators (KPIs) and other business-driven measures, should be added and measured for a model in a production [20].

(d) Deployment

Once the model is trained, tuned, and evaluated, it is ready for prime time. Deploying a model involves taking it from development and turning it into an executable form. Once we train the ML model inside a development tool such as a Jupyter notebook, the trained model disappears once we stop the notebook. To deploy the model into production, we can use tools such as Pickle to save and load the model. Pickle is a serialization and deserialization module.

(e) Model Monitoring

A deployed model must be continuously monitored to ensure that it is performing at a satisfactory level. Monitoring involves keeping track of several metrics and, based on their values, deciding when the model needs an update. It provides insight that facilitates the detection of faults or errors that may occur unpredictably in the production environment. Threshold values are defined for various metrics, and procedures are implemented to trigger alerts when metric values are observed below their respective thresholds [61].

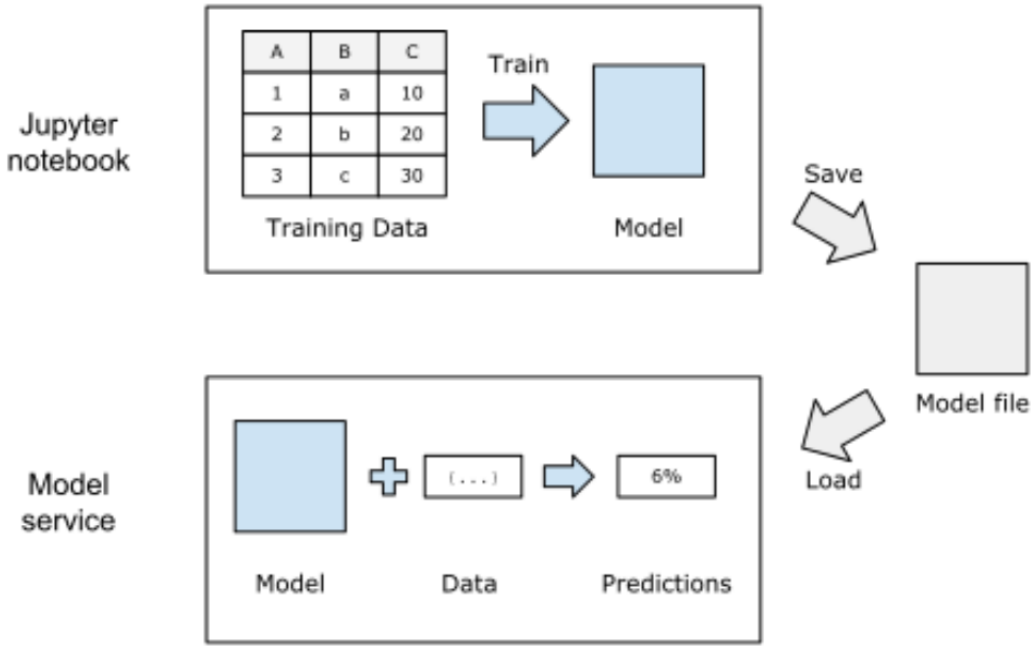

Figure 11: Train a Model, save the model and load it with Pickle

The components of the ML pipeline need to be executed or orchestrated in order for the pipeline to function correctly using pipeline orchestration tools such as Apache Airflow and Kubeflow as shown in Figure 12

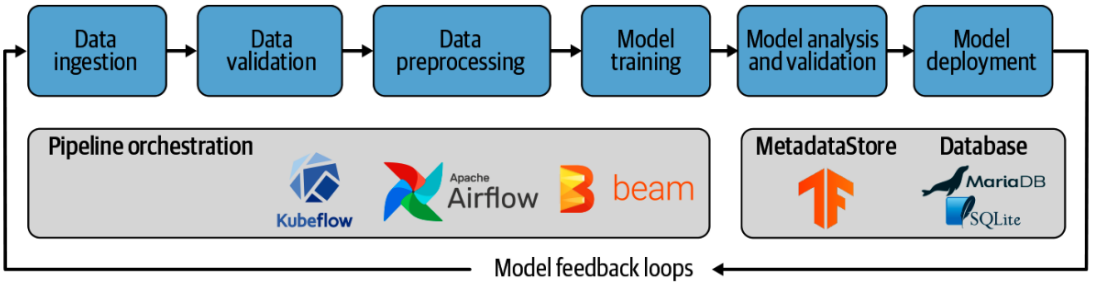

Figure 12: A machine learning pipeline with workflow orchestration tools [2]

2.4.1 Improve Model Performance

There are many ways to improve model performance [69].

(a) **More Data** One of the key strategies to improve model performance is to add more data.

(b) **Feature Engineering** transforms raw observations into desired features with the help of statistical or ML techniques.

(c) **Feature Selection** is the process of walking through the large space of input features and deciding which features we want to keep and determine which features in the dataset we want to analyze further.

(d) **Multiple Algorithms** performance can be improved using multiple algorithms aggregating outputs. A well-known example is Google's search engine [69], which uses many different statistical models for its results.

(e) **Ensembles** It is possible to create multiple models using the same or different algorithms or to generate subsamples of the training data and combine predictions in order to achieve a better-performing model. That is what an ensemble method does.

(f) **Cross-Validation** is a method to evaluate models by training them on subsets of the input data and evaluating them on the complementary subset of the data.

(g) **Hyperparameter Tuning** One of the significant parts of model development is tuning algorithms to spot a good-performing model that is best suited for the task at hand.

Some of the methods for hyperparameter tuning are **Grid Search** and **Randomized Search**. Grid Search performs an exhaustive search on the hyperparameter set specified by users. On the other hand, in randomized search, instead of searching for all hyperparameter sets in the search space, the method evaluates a specific number of hyperparameter sets at random.

2.5 ML Software Quality Fundamentals

Software quality assurance is an approach to detecting faults, exceptions, and runtime errors to improve reliability, and quality, and build user trust in using software systems. Due to the stochastic nature of machine learning, high data dependence, the black-box nature of algorithms, machine learning model prediction uncertainty, and ML systems quality assurance, ML systems presented new challenges not seen in conventional software systems. Significant research efforts and practices have been devoted to understanding the quality assurance challenges when adopting ML applications into software systems to provide a holistic view [17].

(a) Software Engineering Practices

The integration of ML components into large software systems has changed software engineering practices, such as quality assessment, testing, requirement modeling, and implementation [87, 19, 17]. The development of ML is composed

of: Problem definition, data processing, feature engineering, model development, model evaluation, model deployment, and monitoring as depicted in Figure 13. ML employs the inductive development technique, in which knowledge and rules of models and algorithms are derived from the training data. And, software engineers are responsible for writing down programs for the training program.

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software [81]. Researchers [87, 44, 19, 17], examined the SE practices and differentiated ML and conventional software development by interviewing a group of researchers from Amazon, Google, and IBM. Furthermore, they examined the practical challenges of ensuring quality in ML systems and the challenges were due to data, employed practices, and standards [17].

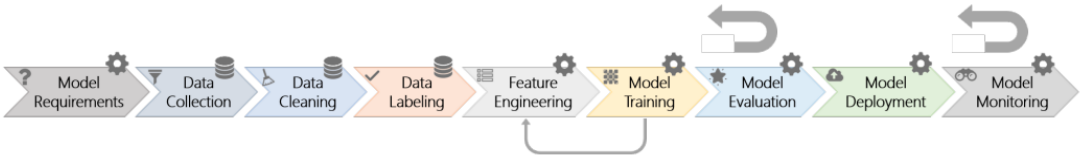

Figure 13: ML development stages used in the study [19]

The researchers addressed the unique challenges of using conventional software development styles for ML.

- **Requirement Collection** ambiguous requirements and a lack of standards are among the top challenges faced by software engineering teams. Collecting complete ML requirements in advance is not achievable.
- **ML Design** The difficulty to understand the internal representation of ML algorithms can cause privacy breaches, performance decays, and fairness issues. These challenges prevent ML designers from having a detailed design and architecture.
- **Implementation** Inadequate data versioning tools, lack of code reuse and modular design, the longer evaluation process, stochastic nature of predictors, data dependency, and lack of well-defined workflow for integration due to rapid ML evolution are among the challenges that make ML implementation complex.
- **Testing** is challenging due to the lack of proper adversarial robustness and increased testing scope to test both data, learning program, and framework testing.

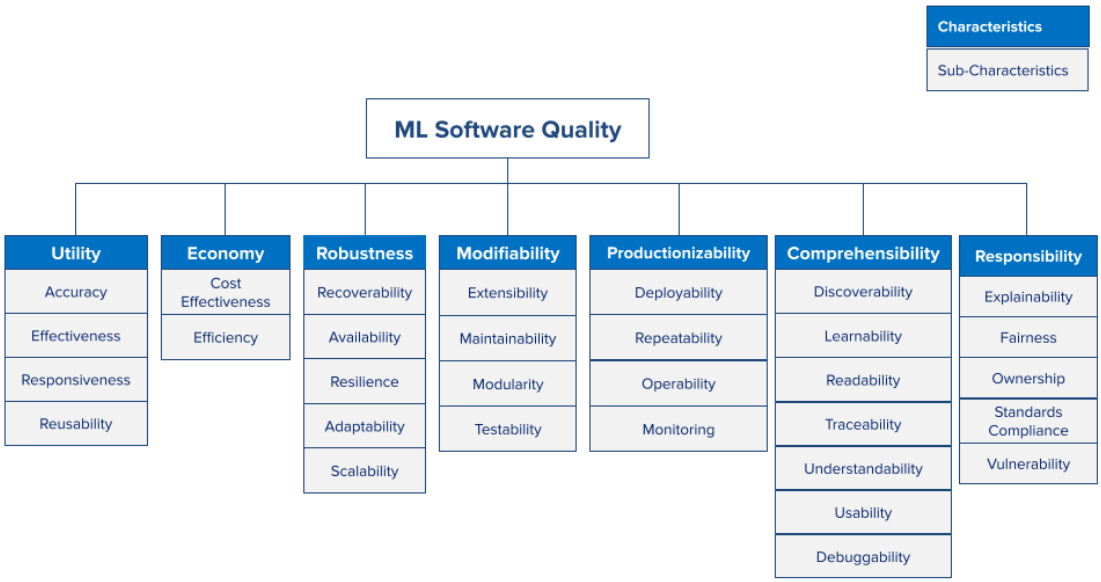

Figure 14: A Software Quality Model for ML Systems

In summary, versioning data are more complex, and model customization and code reuse require different skill sets. Also, it is imperative to develop a solid data pipeline and use data versioning tools capable of continuously loading and managing data.

(b) A Software Quality Model for Machine Learning

Software quality is the capability of a software product to satisfy stated and implied needs when used under specified conditions [37]. The primary goal of ML is to optimize performance, robustness, correctness, model relevance, data security, fairness, etc of the model [37]. The rules are not hard-coded, rather they are detected from data. The quality depends on the input data, ML code, sets of algorithms, and frameworks. Data is collected and pre-processed for use whereas the learning program is the code for running to train the model and frameworks offer developers various algorithms to choose from when writing the learning program. A comprehensive review of the different quality properties and challenges was discussed [92]. The researchers conclude that software quality in ML has not been thoroughly explored and the term quality is not clearly defined. This makes it difficult to have a standard framework to specify and examine different aspects of ML systems [37].

2.5.1 The Dimensions of Software Quality in Machine Learning

The ML community has focused predominantly on improving the performance of ML models. Fair to say that the term quality in the context of ML refers to the robustness, correctness, fairness, etc, for the ML model predictions [73, 92]. Despite the intensive support for building ML applications, it is challenging when it comes to evaluating and assuring their quality. As a result, there is an increasing need for the reliability of such systems. Researchers present a systematic survey of machine learning testing that looks at the different quality properties of supervised learning modeldrawn from various software testing research [92, 73]. These properties are correctness, model relevance, robustness, fairness, efficiency, fairness, and interoperability. Due to the fact that there is no widely accepted standard, these properties were drawn from prominent research works present in ML literature and software engineering professional practices.

The quality assurance of ML systems is the result of multiple coordinated activities such as continuous data quality checks, model training and performance, hyperparameter tuning, writing automated tests, continuous deployment and performance monitoring, data, and model version control, ML framework quality, etc [92, 71]. In summary, the quality of data, and the quality of ML algorithms are the main aspects of an ML model that needs to be tested/quality assured. The key quality components(dimensions) and the unique challenges are discussed next.

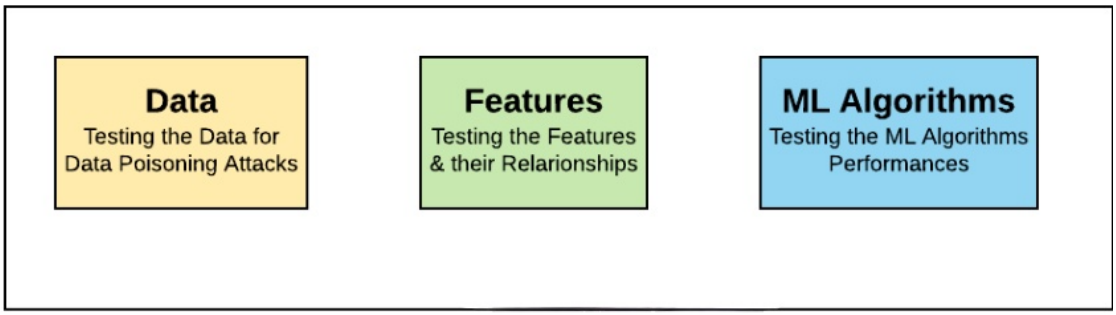

Figure 15: A quality assurance framework for testing ML models [9]

(a) Data Quality

Data is the backbone of ML applications. Real-world datasets are often dirty and until recent times, data quality is not given much thought, and high-quality data is generally taken for granted [92, 73]. In real-world ML applications, poor quality of data is one of the most prevalent issues causing many software defects [77, 25]. Training data is crucial to ensure that the ML system accurately represents and predicts the phenomenon it is claiming to measure[74]. As a result,

practitioners must invest enough time to perform continuous sanity checks for all data sources. The data quality dimension is something that a data item, record, data set, or database can either be measured or assessed. *For example, a test data set can be measured as 97% complete, or the result of an accuracy assessment for a data item in a test data set was 85%.*

(b) Datasheets for Datasets

Datasets are central to the machine learning ecosystem and formulate problems, organize communities, and act as an interface between academia and industry [34]. There is no standardized way to document how and why a dataset was created, what information the data contains, what tasks the data should be used for, and whether the data might raise ethical or legal concerns [38]. A comprehensive study in Google [38], proposed a template for datasets, a tool for documenting the datasets used for training and evaluating machine learning models.

Datasheets for datasets encourage data creators to thoroughly reflect on the data creation process, enabling them to uncover possible sources of bias in their data or unintentional assumptions that they have made[38]. For dataset consumers, the information contained within datasheets can help ensure that the dataset is the right choice for the task at hand [38]. The Datasheet contains questions about dataset motivation, composition, collection, pre-processing, labeling, intended uses, distribution, and maintenance. However, unlike other tools for meta-data extraction, datasheets are not automated yet but are intended to capture information known only to the data creators.

The researchers [38] designed seven guiding questions for creating datasheets.

- **Motivations** explain the motivations for creating the dataset
- **Composition** provide detailed information about the dataset.
- **Collection Process** describe the data collection process
- **Processing** whether the data is labeled and how it was done.
- **Uses** The tasks the dataset is intended to be used for and limitations
- **Distribution** how the dataset will be distributed
- **Maintenance** Who and how the dataset will be maintained

The template was used to create customized datasheets for various tasks.

- **Dataset Card for BookCorpus** A datasheet for a text-to-speech dataset for training a voice recognition model [21].

- **IdenProf** A datasheet for a dataset of images of identifiable professionals or celebrities.
- **Retiring Adult Datasheet** A new dataset for fair machine learning

Definition 2 *Data Cascade is the compounding events causing negative, downstream effects from data issues, that result in technical debt over time [74]*

We illustrate the unique challenges of data cascades, the major causes, and the properties of data cascades in ML systems.

(i) Challenges of Data Cascades

Data is the critical infrastructure necessary to build AI systems, yet dataset work is the least incentivized aspect of AI research and development. Data is often viewed as databases and legal issues. However, for the high-stakes domains of artificial intelligence, data is believed to play a key role in elevating the role of data as a first-class citizen of ML systems. AI researchers [20], conceptualize data cascades, their characteristics, and their impact on the AI life cycle. They derived awareness of the need for urgent structural change in AI research and development to incentivize care for data excellence of common AI types.

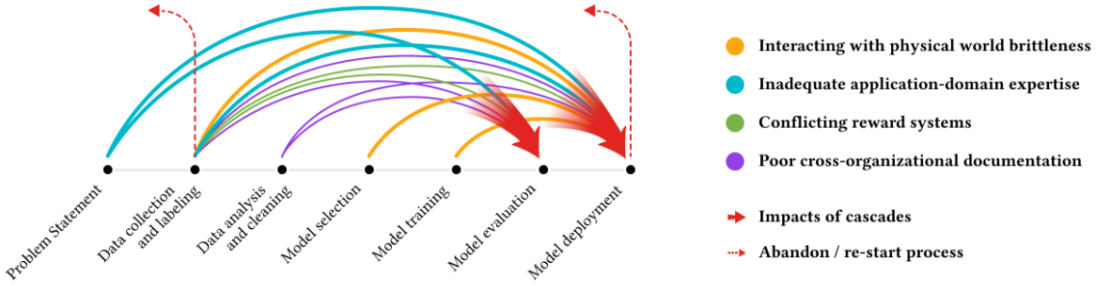

Figure 16: Data cascades in high-stakes AI [74]

(ii) Causes of Data Cascades

The following major causes of data cascades in ML and AI systems were identified [20, 74].

- The Interaction with physical environment brittleness
- The Inadequate application-domain expertise
- The conflicting reward systems
- The poor cross-organizational documentation.

It is a result of the fact that AI systems communicate with undigitised environments. This leads to several reasons for a model to break such as limited training data, complex underlying phenomena, and volatile domains[74].

(iii) Properties of Data Cascades

Data cascades are complex, long-lasting, occur frequently, and possess the following main properties [74, 20].

- They are opaque; there are no clear indicators to detect or measure their effects
- They are triggered when conventional AI practices are applied in high-stakes domains. These are viewing data as operation, moving fast to proof of concept, hacking model performance without consideration for data quality, and undervaluing domain expertise.
- Cascades have negative impacts on the development and deployment process, leading to harm to beneficiary communities, discarding entire datasets, and performing costly iterations
- Cascades are avoidable by step-wise and early interventions in the ML process

(c) Evaluation

Performance metrics such as accuracy have been the most exercised evaluation metrics. The choice of evaluation metrics depends on the application domain, training data, ML task, and algorithms at hand. ML testing approaches have been growing rapidly as rapid performance reduction has become a major challenge in production ML systems [92]. The reduction is due to production ML systems relying on constantly changing data. To deal with this unprecedented online data, practitioners should adopt best practices, such as automating model deployment, monitoring the behavior of deployed models, performing checks to detect skews between models with active dashboards, and logging serving model performance [79]. ML engineers should share clearly defined objectives, capture the training goal in a metric that is easy to measure and explain, properly document features that are not often used in the training, write reusable training scripts, automate hyperparameter and model selection settings, and monitor model quality and performance. Perhaps most importantly to ensure the quality of ML systems, developers should perform automated tests, and model and data versioning.

(d) Security

ML security is a rapidly growing research area because ML models have been found vulnerable to manipulation of behavior, deliberate misclassification, and revealing identities. The large input space makes models inherently vulnerable to security risks. There are no generally accepted standards for effective defenses against machine learning-specific attacks. It is exacerbated by the lack of knowledge of how modern ML models, particularly deep neural networks, actually work [53, 75, 68]. ML security risks can be evaluated using the common attacks on machine learning systems: poisoning attacks, evasion attacks, and model and training data [75].

(e) Privacy

AI systems involve a lot of sensitive information like medical records, financial information, people's moving trajectory, etc. The question is how can we protect sensitive data used by AI systems. And how can we project the models learned from those sensitive data?

(f) Fairness

Human data is used to build AI systems, and humans are naturally biased. AI systems are also biased, and in some cases, they exaggerate human biases. Therefore, the big question is how can we ensure ML models make fair decisions? In particular, how can we ensure they are not making unintended decisions about some sensitive data of individuals such as gender, religion, race, and sexual orientation?

(g) Interpretability and Explainability

One of the main drawbacks of AI systems is the lack of transparency. It is very complicated to explain how AI systems reach decisions. What is trending is how to keep ML models explainable and interpretable so that we can understand their decisions and patterns. Local Interpretable Model-agnostic Explanations (LIME) is a visualization technique that helps explain individual predictions.

(h) Verification of AI Systems

AI systems are very complex. Their implementation and configuration are very difficult to assess. So the question is how can we prove the ML models work as they are expected to behave? The goal of verification of AI systems is to verify if the prediction returned by ML models is correct and verifiable.

(i) Model Key Performance Indicators (KPIs)

Prediction is a technique for estimating how well a model performs on unseen data, in terms of performance metrics such as accuracy. However, these metrics evaluate a model in isolation, independent of the context of how the model is being used [20]. ML models are rarely exposed to end-users. Typically, ML models are embedded within an application and the application leverages the model to help make key business decisions, such as whether to approve or reject a loan [20].

End-users do not care about performance metrics. What ultimately matters is how well the application is performing as visible to the end-user [20]. User-visible application performance is generally tracked with business metrics called key performance indicators (KPI), for example, sales rate, click rate, customer satisfaction, time on page, etc. Thus, when evaluating models in the context of an application, performance metrics are not sufficient. Instead, it is recommended to critically understand how the model behavior is impacting the application's business metrics. Failure to do so may result in wasting resources on improving model metrics that have little or no impact on what matters to end-users.

(j) Model Cards for Model Reporting

There are no standardized documentation procedures to communicate the performance characteristics of trained ML models. This lack of documentation is especially problematic when models are used in applications that seriously impact people's live [59]. Model cards are short documents accompanying trained machine learning models that provide important information about the model, like details of how it was trained and evaluated, where it can and cannot be used, its context of use, etc [59]. It enables the standardization of knowledge for efficient analysis by model validators to increase transparency by communicating key information.

Model cards are important for ML practitioners to better understand how well the model might work for the intended use cases and track its performance over time, policymakers can understand how a machine learning system fails or succeeds in ways that impact people, and model developers can compare the model's results to other models in the same space, and make decisions about training their own system. A typical model card contains model details, the intended use of the model, evaluation metrics and data, ethical considerations of the model, etc.

2.6 Technical Debt in Machine Learning Systems

When software engineers adopt shortcuts that are not in line with engineering best practices, they could accumulate hidden costs. This accumulated hidden cost is known as technical debt and compounds silently in contrast to the loud software failures in conventional software. Technical debt can be difficult to detect as it exists at the system level rather than the code level.

Definition 3 (Technical Debt) *When software engineers prioritize the speed of prototype deployment over all other factors in development, the build-up of these ongoing costs is referred to as Technical Debt [77]*

Hidden Technical Debt in Machine Learning Systems is the most comprehensive paper exploring the source of technical debt in machine learning systems. The increased complexity of ML system makes them challenging to develop, maintain and improve, resulting in a strong entanglement within the system. It is due to the fact that each feature and parameter of the input are interconnected and influence each other. Also, the use of general-purpose machine learning frameworks that often are not directly compatible with each other is another cause of technical debt. Consequently, a large amount of data transformation code needs to be written to achieve framework interoperability [77].

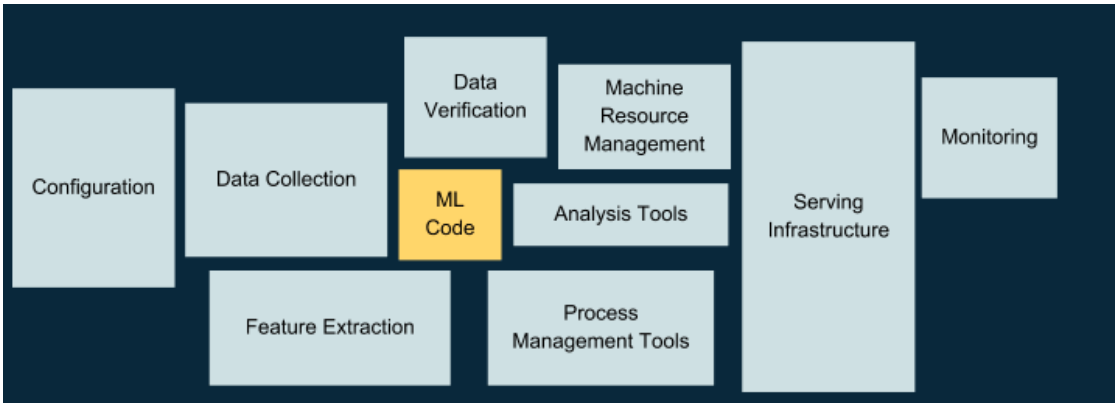

Figure 17: A small fraction of real-world ML systems is composed of ML code [77]

The ML complexity emerges from the various components shown in Figure 17 to execute the system successfully. The key sources of technical debt [77, 76] in ML systems are the following

- **Unstable Data Dependencies** ML input data are unstable, and they change qualitatively and quantitatively over time.

- **Feedback Loops** model output influences the training data for future versions of the model.
- **Entanglement** ML systems that use multiple algorithms where downstream models consume outputs of other upstream models.
- **Monitoring debt** Evaluating a model on a static dataset is easy, but building a monitoring infrastructure to evaluate model quality in production requires substantial cost.
- **Reproducibility debt** It is difficult to reproduce prediction because of the randomized algorithm parameter initialization, data-dependent decision logic, and continuous interactions with the external world.
- **Lack Pipeline automation** The lack of a robust pipeline to automate and monitor steps, remove dead code, write usable and well-documented code, and avoid clones. This might save some effort initially, but contribute to increased debugging and release efforts in the future.

Technical debts are primarily caused by low-quality software design decisions, relying on quick and risky fixes instead of complete refactoring, inadequate testing, poor understanding of ML framework architecture, and last-minute changes to specifications. There are helpful metrics that help assess the level of technical debt such as comparing newly discovered bugs vs. closed bugs, code quality metrics, code coverage, and technical debt ratio.

2.7 Bugs in Machine Learning Systems

A software bug refers to an imperfection or unintended behavior in a software program that causes a discordance between the existing and the required conditions [92]. An ML bug refers to any imperfection in ML that causes a discordance between the existing and the required conditions [92]. Errors, faults, exceptions, and failures are the sources of unintended behavior in software systems. A fault is an incorrect step, process, logic, or data definition in software programs. Failure refers to the inability of a system or component to perform its required functions within specified performance requirements and is one type of fault [73]. Faults arise from mistakes and errors made by software engineers, for example, wrong instruction in a line of code.

ML testing refers to any set of activities designed to reveal bugs. ML bugs and ML testing point to three characteristics of ML systems: The testing properties,

the testing components, and the testing activities. ML systems have a set of properties that require testing, such as correctness, robustness, data privacy, fairness, etc. ML testing workflow is a set of activities, such as test input generation, test oracle identification, test adequacy evaluation, and bug report analysis.

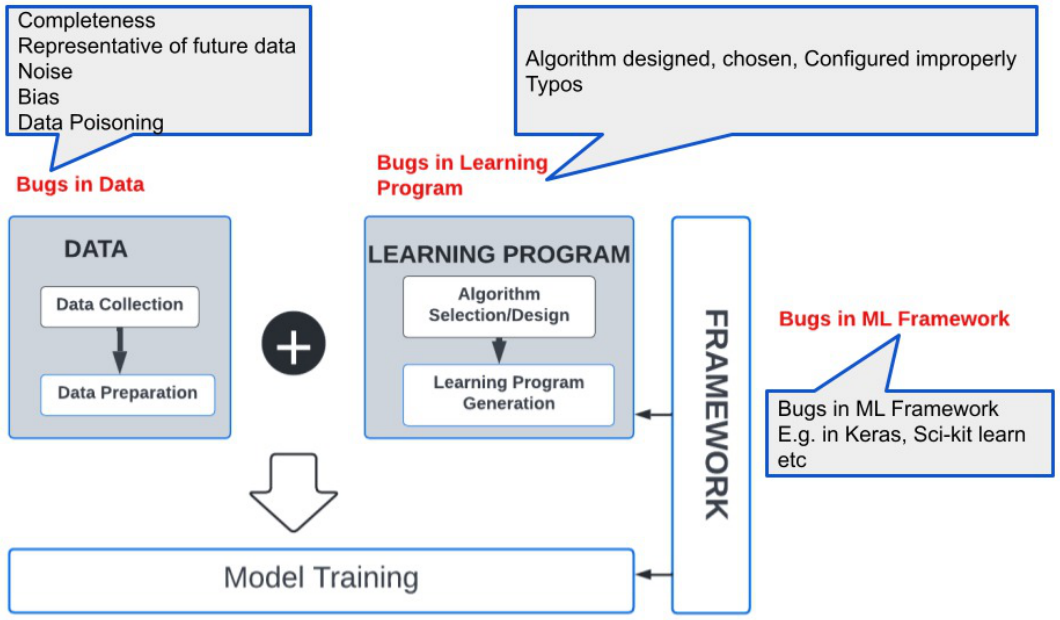

Figure 18: Machine learning components where bugs can reside [92]

Bugs in ML systems can exist in the data, the learning program, and the framework as shown in Figure 18.

- **Bugs in Data** are bugs in data that affect the quality of the ML model and can be amplified to yield more serious issues over a period of time because of the data cascade problem [92, 74].
- **Learning Program Bugs** are bugs in the learning programs that arise due to the algorithm being designed, chosen, or configured improperly.
- **Bug in the Framework** frameworks offer algorithms to help write the learning program, and platforms to help train ML models and predict and validate models. However, like any other software program, frameworks are prone to bugs as well.

2.7.1 Machine Learning Bug Data Analysis

ML has become an integral component of software applications, however, practitioners are not yet fully aware of the challenges facing them. We explored the root causes and impacts of frequently occurring bugs in machine learning systems in order to gain deeper insight into them. ML discussions on Stack Overflow [46] performed a thorough analysis of 3243 highly-rated posts related to 10 popular ML libraries, including but not limited to Scikit-Learn, Keras, TensorFlow, and Spark ML. They also discussed DL bug properties using 2716 high-quality posts on Stack Overflow and 500 GitHub bug fix commits from 5 DL systems, namely, Caffe, Keras, Tensorflow, Theano, and Torch [45]. Furthermore, [42] studied more than 1000 artifacts of GitHub commits of TensorFlow, Keras, and PyTorch.

They analyzed only the highly-rated questions from reputable users. On Stack Overflow, each question is rated by the community where the score of a question is computed as the difference between positive and negative votes. The higher the score the better the quality of the post. Posts with a score of 5 or above and from reputed users were selected for analysis.

(a) The Common Types of Bugs

Manual analysis of posts from Stack overflow and GitHub fix commits results in a taxonomy of bugs: Data preparation, Modeling, Training, Evaluation, Hyperparameter Tuning, and Prediction [45, 46]. A similar study of faults focused on DL systems produced a high-level bug taxonomy: Model, Training, API usage, GPU usage, and Inputs and Tensors [42]. Thus, the frequently-occurring bugs are:

- **Coding Bugs** refers to bugs originating from a coding syntax problem.
- **Data Bugs** arise if an input to a given machine learning model is not properly handled (e.g., type or shape mismatch).
- **API Bugs** It is common for API bugs to be inherited without the developer's knowledge. API bugs are related to API evolution and usage.
- **Structural Bugs** these bugs are the result of incorrect definitions of the learning model structure such as a mismatch of dimensions between different layers of deep learning models.

(b) Root Causes of Frequent Bugs

Based on the description of Stack Overflow posts and GitHub fix commits [46], the following are the root causes for many machine learning bugs.

- **Incorrect model parameters** resulting in modeling errors caused by inappropriate parameters
- **Confusion with the Computation Model** bugs are often caused when users are unfamiliar with the underlying computation model.
- **The Absence of Inter API Compatibility** inconsistencies when using multiple frameworks or API versions to develop applications.
- **The Absence of Type Checking** type-checking bugs occur when API calls are made using the wrong type and the number of parameters.
- **API Changes** The reason for API change bugs is the release of an updated version of ML libraries, however, practitioners use the old API coding style.

(c) Impact of Machine Learning Bugs

The various bugs discussed above can result in poor performance, program crashes, incorrect functionality, and memory overflow on the ML program.

- **Bad performance** is one of the noticeable effects of ML bugs.
- **Program crash** is the most severe effect of bugs and happens frequently
- **Incorrect functionality** occurs when the runtime behavior of the ML systems lead to an unexplainable outcome, for example, an incorrect output format.
- **Memory out of bounds** ML systems are data intensive and often halt due to inadequate memory resources to train or predict a particular mode generates

In summary, the researchers [45, 46] identified the most challenging stage as Model creation and data operations. As for the ML pipeline stages, it was determined that when it comes to libraries that support ML clusters such as Mahout and Spark ML, the model creation stage presents the greatest difficulty, followed by data preparation. In addition, they found that type mismatches were present in most libraries, and shape mismatches were common in DL libraries. Another finding was that the problems with modeling were consistent throughout the study period.

3 Testing Techniques for Machine Learning Systems

According to the IEEE Standard Glossary of Software Engineering Terminology, software testing is described as an activity in which a system or component is executed under specified conditions, the results are observed, and an evaluation is made of some aspect of the system or environment [12]. A system under test represents a test component that is ready for validation for its correct operation, often called the software product. Test cases are a set of test inputs, execution conditions, and expected results developed for a particular objective such as to verify compliance with a specific requirement or regulation [12, 6]. A collection of the test cases used for the test execution process of the software application is called a test suite.

The granularity of the entity to be tested is defined by the test level. Conventional software unit tests, integration, and system-level testing can be used in ML systems, but they may require further specialization. Therefore, there are input tests, model tests, integration tests, and system tests in machine learning [47, 73]. The various testing levels in ML systems are mentioned in **??**.
The following are the testing levels or granularity in ML systems.

- **Input Testing** analyzes the training and test data used to evaluate the model.
- **Model Testing** aims at finding faults due to suboptimal model architectures, training processes, or wrong hyperparameters tuning strategy.
- **Integration testing** examines issues that arise when multiple models and components are integrated, even if each component or model behaves correctly in isolation.
- **System testing** aims to test the ML system as a whole in the environment in which it is supposed to operate.

The core differences between conventional software testing which has relatively stable and clear test oracles, and testing ML which needs to deal with continuous

changing behavior with bugs that exist not just in the algorithm but also in the training dataset and even in the ML framework emerged from the fundamental difference of ML systems [92, 31].

3.1 The Oracle Problem

Testing involves examining the behavior of a system in order to discover potential faults. Given an input for a system, the challenge of distinguishing the corresponding desired, correct behavior from potentially incorrect behavior is called the test oracle problem or simply test oracle [22, 92]. Test Oracle provides testers with reliable resources for conducting testing. Test oracles can be classified into specified test oracles, derived test oracles, implicit test oracles, and human test oracles [22]. Test oracle identification using human testers involves rerunning old tests with a modified version of the program that increases test execution, which takes extra time and resources.

Reducing the involvement of human testers as a result of automating test oracles is one of the biggest challenges. There was a time when the oracle problem was reserved only for human experts, where they had responsibility for examining the behavior of the system. The human experts apply knowledge and previous experience gained in solving similar problems [22]. The main challenge with the manual oracle problem is the Human Oracle Cost, which is the effort invested by human experts to create test cases and/or evaluate them.

Quantitative and qualitative cost reduction techniques can be used to lower human resource costs [22]. Quantitative cost reduction focuses on reducing the amount of work that the tester has to do for the same amount of test coverage. This reduces the number of test cases and test suites in return. On the other hand, qualitative cost reduction focuses on reducing the amount of work needed to understand and evaluate test cases [22]. The qualitative human oracle cost reduction relies on the incorporation of human knowledge for a better understanding of test cases through the simulation of knowledge and experience from the human expert. This means that using automated oracles that rely on script execution is not possible.

3.2 Machine Learning Testing Challenges

ML has made great strides, delivering human-level performance and these advances have led to the adoption of machine learning in safety-critical systems such as self-driving cars, malware detection, unmanned aerial vehicles, fraud detection, and heart failure detection [73, 92]. Some of the recent dangerous accidents associated with such systems, for example, a self-driving Google car crashed into a bus in California in February 2016 and a Tesla car crashed into a trailer in Paris in December 2021 without being recognized as an obstacle [92, 73]. In addition, nearly 400 crashes by self-driving cars in the US caused a lot of damage as reported by Al Jazeera.

Many of problems of deployed AI and ML emerges from lack of proper testing. ML systems have been classified as "non-testable systems" [73, 92, 36, 54, 48], because testing sucy systems poses new challenges for the software testing community [92, 92, 56]. First, ML systems require more complex requirements but that does not mean that the need for traditional software testing disappears. Second, these systems unlike conventional systems lack an explicit oracle. Third, machine learning systems are composed of several components that deal only with traditional software, and are not affected by machine learning algorithms. Thus, they cannot be fully tested with existing software testing techniques which rely on explicit oracle and program control flow.

Machine learning testing refers to any activities designed to reveal bugs [92]. The correct use of model evaluation, model selection, and algorithm selection techniques is vital for machine learning [71]. Among the recommended subset of techniques to be used to address different aspects of model selection and evaluation are; understanding model evaluation metrics, maintaining the bias-variance tradeoff, hyperparameter optimization, model parameter understanding, applying cross-validation techniques, considering ensemble methods, algorithm comparison, understanding essential model evaluation terms and techniques, and data exploration to better understand the business problem.

There is an indisputable distinction between ML evaluation and ML testing. ML evaluation evaluates the performance of a model on unseen data using score metrics [90]. However, ML testing involves checking the behavior of models to reveal bugs and ensure that the learned logic is working as intended. The unparalleled challenges associated with testing ML systems stem from the non-determinism nature and blox-box decision-making of ML algorithms. Researchers [55, 84, 92], identified the following unique challenges

- **Absence of Test Oracles** ML outputs do not have defined expected values

against which actual values can be compared.

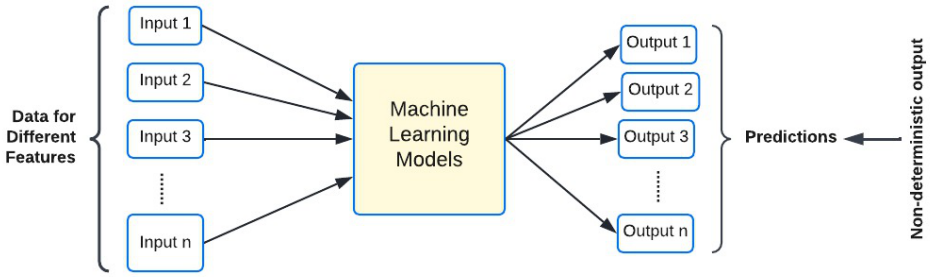

Figure 19: The oracle challenge is to verify the effectiveness of model predictions against the expected values [84]

- **Large Input Space** It is difficult to pick the right set of input data and their combinations from a large input space to trigger ML faults in testing.
- **Algorithmic Uncertainty** arises from a lack of complete specifications of system requirements and random initialization of algorithms.
- **Lack of a Testable Specification** The black box testing tests against specified requirements, however, ML is intended to provide generalized behavior, while a requirement is meant to document specific behavior
- **Train Test Split** The random split API produces less representative data. Model reproducibility can be impacted by the random and non-representative split.
- **ML is less obvious to Modularize** poor performance can be arising from the training data, the learning program, and even the learning framework.

3.3 Machine Learning Testing Workflow

ML bugs and ML testing touches three aspects of ML systems. These are the required conditions, the machine learning items, and the testing workflows/activities. There are several activities involved in ML system testing as shown inFigure 20.

(a) **Test Input Generation** to generate appropriate test inputs from a large input space. DeepXplore and DeepTest are the widely used tools for generating test cases. According to [41], synthetic data with repeating values, or extreme values can be generated to test the learning program and generate test inputs.

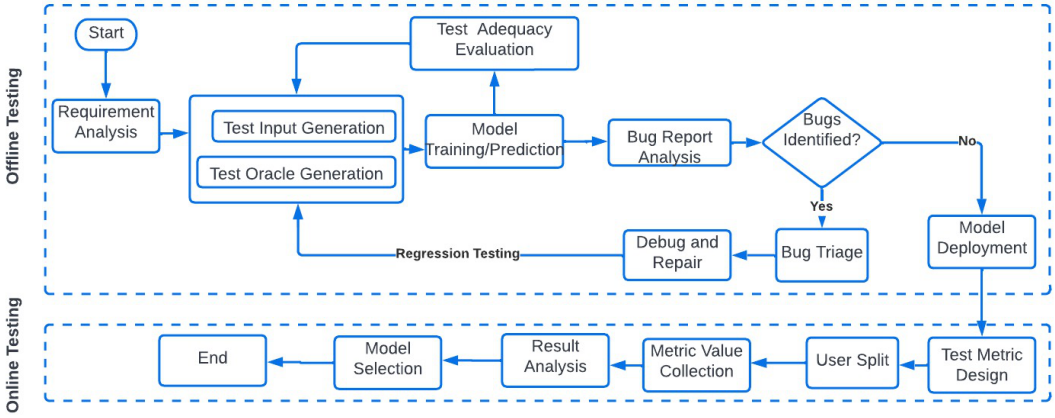

Figure 20: Machine learning testing workflow [92]

(b) **Test oracle identification** The key problems when testing ML system is the identification of test oracles. Metamorphic relations based on various data transformations are mostly used to generate test cases in the case of ML testing [92, 92, 36, 22].

(c) **Test Adequacy Evaluation** Code coverage for ML testing is not a demanding criterion as the decision logic is determined by the training data. Coverage methods such as mutation testing can be used as test adequacy evaluation metric.

(d) **Test Oracle Prioritization** to reduce the computation cost of generating test inputs from a large input space, we need to prioritize the inputs.

(e) **Bug Report Analysis** by analyzing bugs, we can understand and identify bug frequency, bug categories, bug root cause, bug severity, and bug-fix time [92, 36, 45, 46].

(f) **Debug and Repair** debugging is useful to construct a new program with improved performance, but with similar semantics to the original program [92]. For example, DeepXplore improved classification accuracy by 3% and DeepTest boosted autonomous cars deep learning model accuracy by 46%.

(g) **Regression Testing** is used to fix a defect when a new feature is added, when changes are made to an existing feature, and to maintain the software product bug-free [86].

3.4 ML Systems Testing and Validation Techniques

AI and ML researchers [78, 30, 93, 41, 35], have discussed the challenges associated when testing ML, particularly the oracle problems in a great detail. A learning algorithm might be bug-free but still, deviate from the desired behavior due to faults during the training process. ML considers the data to be the oracle since the ML model heavily relies on it. Bugs in data affect the quality of the generated model and can be amplified to yield more serious problems over a period of time. The authors suggested that the components in ML systems must be verified.

3.4.1 Data and Model Validation Tools

The ML community has focused on improving the accuracy of the model, less attention is paid to the equally significant problem of monitoring the quality of data. Errors within the input data can nullify any gains in accuracy. Data cleaning and feature engineering are common best practices. However, developers are not always aware of the best practices for transforming data which can lead to a poor and less robust model. Practitioners introduced tools to validate the input data and evaluate the model.

(a) DeepChecks: Model and Data Validation Library

A DeepChecks is an open-source Python framework for testing ML Models and Data [32, 16]. They are checks for different phases in the ML workflow as shown in Figure 21

- Data integrity checks
- Train-test validation (distribution, drift, and methodology checks)
- Model performance evaluation

DeepChecks introduces three important terms to handle the tests: Check, Condition and Suite. A check aims to inspect a specific aspect of the data or model. Checks enable a user to inspect a specific aspect of the data and models. A Condition is a function or attribute that can be added to a Check. Essentially it contains a predefined parameter that can return a pass, fail, or warning result. A Suite is a module containing a collection of checks for data and models.

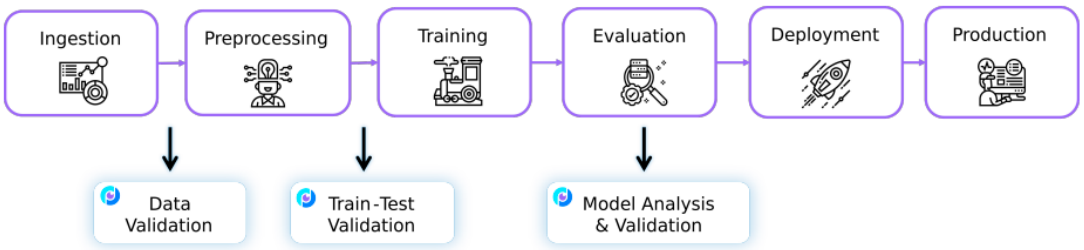

Figure 21: The schema shows three different tests that could be performed in an ML pipeline [32]

(b) Drifter-ML

Drifter ML is a model and dataset testing tool primarily written for models developed using the Scikit-Learn system. The program consists of five modules. These are

- **Classification tests** to test classification models.
- **Regression tests** to test regression models
- **Structural tests** to test clustering algorithms.
- **Time Series tests** to test model drifts
- **Columnar tests** to test tabular datasets.

Drifter-ML is specifically written for Scikit-learn. Thus, all classes and methods are synced with Scikit-learn, making data and model testing easy.

(c) Kolena: MLOps Platform for Building Better ML Models

Kolena is a Python-based framework for ML testing. The tool also includes an online platform where the results and insights can be logged. Kolena specializes in automating the ML unit testing and validation process at scale.

Kolena argues that splitting the datasets into train and test provides a global representation of the entire population distribution. However, it fails to capture the local representations at a granular level. This leads to the failure of the model in the real world.

(d) Data Linter: Sanity Checking for ML Data Sets

The Data Linter is a tool to inspects ML data and identify potential issues and suggests useful feature transforms for a given model [43]. The tool was evaluated on a broader set of data from Kaggle with 600 publicly available CSV data sets

and it can be useful for improving model quality using specific feature transforms for neural networks, linear models, and SVMs [43].

Specific linting services for various data quality issues include miscoding data Linters to identify data that should be transformed for the model to learn from. Outlier and scaling data linters attempts to detect likely outliers and scaling issues in data, and unnormalized features. Finally, packaging error data linters identify problems with the organization of the data, for example, when there are duplicate and empty examples.

3.5 Smoke Testing

Smoke tests assert that the most crucial functions of a program work, however, do not bother with finer details [41]. The interpretation of smoke tests in ML settings is that the most crucial functions do not crash. For example, ML classification and regression algorithms have two key functions that must always work, given valid input, the training of the classification model, and the prediction for instances given the trained model. Similarly, clustering algorithms have one crucial function, the determination of the number of clusters.

The most fundamental criterion for smoke testing is that implementations must be able to handle common data distributions correctly. More importantly, for effective smoke testing, data should also be chosen in such a way, that it is likely to reveal bugs and crash the software [41].

To perform smoke testing in ML, test inputs are generated using the fundamental notion of equivalence classes. Most ML algorithms work with numeric data if we exclude invalid values such as NaN and Inf. Also, ML algorithms work with categorical data with valid categories. Boundary value analysis and corner cases are used to design test cases from the equivalent classes. Technically speaking, all smoke tests are defined over the same two equivalence classes, i.e., valid numeric and categorical data with valid categories [41]. Equivalent class analysis helps to find inputs that reveal bugs and twenty-two equivalent classes are used to generate input automatically.

3.6 Metamorphic Testing

Metamorphic testing is used to avoid the Oracle problem. This method is based on testing the input-output relationship of the system for multiple iterations. Metamorphic testing is founded on the principle that if the correct output for the input is not known, we can validate or verify the system based on the outputs of multiple related inputs. Metamorphic Testing(MT) [36, 93, 48, 29], is designed as a general ML systems testing technique where existing test cases, particularly those that have not revealed any failure, are used as bases for creating follow-up test cases intended to find uncovered flaws. A test case is transformed into a follow-up test case for which the exact test outcome isn't known, but where the relationship between the source and the follow-up test case is known. Through the execution of the follow-up test case, it is possible to confirm whether the program under test behaves as described in the test suites. Violation of the relation indicates a failure in the system.

ML classification models are typical oracle problems because they can only be specified with stochastic behaviors [22]. Since the expected class of any new data sample is unknown, these samples cannot be used for testing trained models. Transformations over the data samples are available which do not change the unknown class. The transformation is often called metamorphic relations.

3.6.1 Metamorphic Relations

Metamorphic relations are mathematical relations between changes in software input and output over several program executions [92, 73, 36]. They are pseudo-oracles adopted to mitigate the oracle problem. Applying MT in practice requires solving two problems: the metamorphic relation identification problem, also known as the test oracle identification problem, and the metamorphic relation selection problem [22, 36]. Metamorphic relation identification occurs when trying to identify the necessary transformation for a specific problem. Finding such relations may be difficult when there are no obvious symmetries in the input and output data. The test oracle prioritization occurs when several metamorphic relations have been identified, but determining which ones are best suited to discover faults in the system under test is difficult.

Some of the metamorphic relations that we anticipate all classification algorithms to exhibit include: permutation of class labels and features, the addition of informative and uninformative features, the addition of training samples, linear scaling of the test features, Insert inputs with extremely large values, adding all features with zero values, insert data with many categorical data, and add only

single class data for a binary classifier [36, 41, 89, 30, 62, 78]. The metamorphic relations are helpful for revealing bugs in ML image classification applications. We used and evaluated some of the transformations using Scikit-Learn classifiers and four metamorphic relations on a given image data as defined in Table 5 and we discovered a few critical bugs as reported in Table 6

(a) Metamorphic Testing for ML Models

Testing ML models from a quality assurance perspective is different from evaluating machine learning models from a performance perspective. In the case of ML models, each output is singular and can't be verified against so-called actual values, so, some mechanism is required to verify the output of the model. MT comes into the picture. ML models can be tested using MT. This consists of test cases based on metamorphic relations, which could be derived from the specific behavior of the model and data set used.

From Figure 22, a property of the model such as age relates inputs with outputs of the model. The output of the model, Y, is the likelihood estimate of a person suffering from a disease and the relevant input parameter is age, X. In reality, as age increases ↑ and given other key features, the likelihood of a person suffering from the disease increases ↑. There is a metamorphic relation between age and output (the likelihood of contracting a disease).

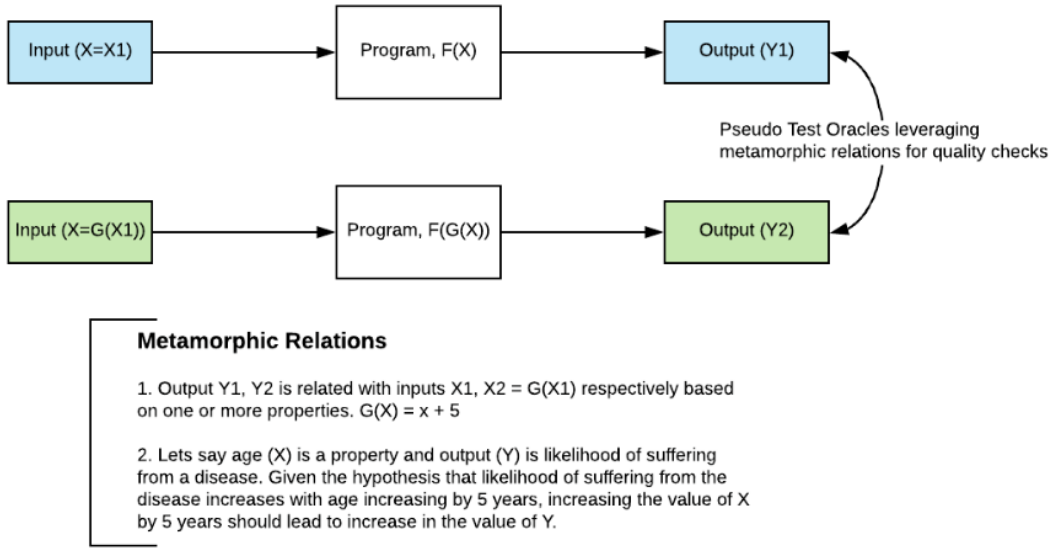

Figure 22: Metamorphic testing of ML models with metamorphic relations [13]

Given the example in Figure 22, the set of test cases using metamorphic relations can be derived(planned) in the following manner:

METAMORPHIC TEST CASES FOR ML MODEL

- Let's For age as **X1** = 30, the output (likelihood estimate as **Y1**) is **0.35**
- For age **X2** = 50, the value of **Y2** should be greater than **Y1**.
- For age **X3** = 40, the value of **Y3** should be more than **Y1** and less than **Y2**
- For age **X4** = 20, the value of **Y4** should be less than **Y1**

(b) Metamorphic Relations for Autonomous Cars

Metamorphic testing can be applied to autonomous driving to cover a wide range of problem spaces, like capturing image variations and LIDAR transformations to detect changes that can happen due to weather and other environmental factors [49].

DeepTest is an automated testing framework for autonomous cars by performing transformations on real images collected from cameras of driving cars to produce transformed images as shown Figure 23. Some of the transformations are: moving, blurring, scaling, zooming, adding fog, adding rain, etc. on the original images. Thus, the autonomous driving model should show minimal divergence on the same input images after the transformations. This way we can say the model is robust to the perturbed data.

Figure 23: Transformations that ML algorithms in an autonomous car need to consider [49]

3.7 Model Behavioral Testing

ML model evaluation covers metrics and summarizes performance on test data. In contrast, model testing involves explicit checks for behaviors that we expect the model to behave after training using the training set and some other model parameters [47]. Model testing helps to pinpoint a bug or area of concern that might cause the prediction capability of the model to degrade. The idea is to build and deploy a robust model that can efficiently handle uncertain data entries and anomalies.

3.7.1 A State-of-the-Art Solution for Testing Models

Model testing can be performed by implementing pre-train and post-train test cases.

(a) Model Pre-Train Tests

These tests include assertions about the dataset as the quality of data that is used for training determines the quality of predictions. That is how data validation in Machine Learning is imperative, not optional! These tests are performed to catch bugs before running the model. They do not require training parameters to run. For example, they can check to see if the training and validation datasets have any missing labels.

Model Post-Train Tests

- Check the shape of the model output and ensure it aligns with the labels in the dataset
- Check the output ranges and ensure it aligns with the expectations.
  - For example, the output of a classification model should be a distribution with class probabilities that sum to **1**
- Write assertions on the training datasets
- Check for label leakage between training, test, and validation data.

(b) Post Train Tests

Model post-train tests are performed on a trained model to investigate the logic behind the algorithm and see whether there are any bugs there. The three types of tests we need to perform on a trained model are the Minimum functionality test, the Invariant test, and the Directional expectation test. These tests are used to analyze a trained model's behavior. For example, predicting the survival chance of Titanic passengers by changing relevant and less relevant features of the Titanic data [91].

Model Post Train Tests

Check for **Invariance** by keeping other features constant and tweaking one feature at a time.

- We do not expect the survival probability to change due to a change in the passenger's name or ticket number

On the other hand, we have **directional expectations** such as:

- Females have a higher survival probability than males
- Higher passenger class having higher survival probability
- Higher fare having higher survival probability

For the directional expectation, changing the passenger's gender and class, we expect the model prediction to react in a specific manner(↑ - ↓).

3.7.2 Behavioral Testing of NLP Models With CheckList

A model that performs well in an offline evaluation but poorly in an online evaluation means that the model has not detected patterns that allow it to generalize broadly enough to understand other data which may look differently. In addition, ML performance metrics such as classification accuracy have been the most widely used way to assess the generalization of classification models. However, accuracy can overestimate the performance of ML models such as NLP models. In order to ensure that the learned behavior is as expected, an ML model must be tested against certain capabilities. This type of testing is called post-train tests, i.e. models need to be evaluated based on individual tasks. Often called model behavioral testing.

Behavioral testing, also known as black-box testing, is a method where we test a piece of software based on its expected input and output. We don't need access to the actual implementation details [28, 72]. For example, let's say we have a function that adds two numbers together and evaluate the add function by writing tests to compare its output to the expected answer.

```
1 def add(a, b):
2     return a + b
3 def test_add():
4     assert add(1, 2) == 3
5     assert add(1, 0) == 1
6     assert add(-1, 1) == 0
7     assert add(-1, -1) == -2
```

Even a simple function such as addition should satisfy certain capabilities. For example, the addition of a number with zero should result in the original number.

Capability	Function Signature	Output	Expected	Test Passed
Two Positive Numbers	add(1, 2)	3	3	Yes
No Change with Zero	add(1, 0)	1	1	Yes
Opposite Numbers	add(-1, 1)	0	0	Yes
Two Negative Number	add(-1, -1)	-2	-2	Yes
			Pass Rate	4/4 = 100%

Figure 24: Capabilities that the add function should satisfy [28]

"Beyond Accuracy: Behavioral Testing of NLP Models with CheckList" was developed as a task-agnostic method for testing NLP models using the principles of behavioral testing. A CheckList is an open-source framework for testing NLP models for linguistic capabilities. The CheckList encourages users to examine how different natural language capabilities manifest on the task at hand and to create tests and evaluate models that represent each capability. Depending on the NLP model, the capabilities might include vocabulary and part of speech, named entity recognition, robustness, negation, and temporal relationships.

There are 3 types of tests proposed in the CheckList framework.

(i) **Minimum Functionality tests (MFT)** Inspired by unit tests in software engineering, it is a collection of simple examples to check a behavior within a capability.

(ii) **Invariance Test(INV)** In this test, we perturb existing training examples in a way that the label should not change.

(iii) **Directional Expectation Tests (DIR)** This test is similar to the invariance test but here we expect the model prediction to change after perturbation.

The authors provide a lot of examples to help us build a mental model of how to test for new capabilities relevant to the task and domain.

(a) **Named Entity Recognition** tests the capability of the model to understand named entities

(b) **Negation** tests how a model understands negation.

(c) **Robustness** evaluates(tests) how a model reacts to small changes to the input data

(d) **Fairness** tests if the model reflects any form of bias toward a demographic data

(e) **Taxonomy**to test how the model understands synonyms and antonyms and how they affect model output.

(f) **Temporal relationship** how a model understand order of events

Figure 25: Examples to illustrate (**1**) Minimum functionality tests (**2**) Invariant tests tests, and (**3**) Directional expectation tests [28]

The library allows users to generate large numbers of test cases, including crafted templates, lexicons, perturbations, and context-aware suggestions, by utilizing a robustly optimized method for pre-training NLP models called **RoBERTa**. In

summary, we can use CheckList to understand NLP models beyond the leaderboard. For example, sentiment classifier models should be invariant to the names of people. Road segmentation models should work regardless of weather conditions, and Fraud probability on product reviews shouldn't depend on customer gender. We can specify the capabilities of a particular model. In an image recognition model, the capabilities might include rotating an object, modifying perspective, adjusting lighting conditions, and changing weather conditions.

3.8 Best Practices for Machine Learning Operations

ML models must go through many experimentation cycles before they can be put into production. There are a plethora of tools and best practices to operationalize modern ML applications.

(a) **Docker and Kubernetes:** Kubernetes and Docker are two well-established technologies in software system operationalization. Docker is helpful for containerizing ML applications, and Kubernetes automates the deployment, scaling, and management of containerized ML applications

(b) **ML Lifecycle Management:** MLflow is an open-source platform for managing the end-to-end machine learning life cycle, including experimentation, reproducibility, deployment, and a central model registry.

(c) **Model and Data Validation:** DeepChecks is an open-source Python framework for testing ML Models and Data. Drifter-ML is another open-source model testing tool specifically written for the Scikit-learn library.

(d) **Databricks Machine Learning** is an environment that makes it easy to build, train, evaluate, and deploy ML learning models at scale. It integrates with popular open-source libraries and with the MLflow API to support the end-to-end ML lifecycle from data preparation to deployment.

(e) **Data and Pipeline Versioning:** Data Version Control (DVC) is an open-source version control system for data, models, and experiments to make models shareable, experiments reproducible, and to track versions of models, data, and pipelines.

(f) **Hyperparameter Tuning:** Optuna is an automatic hyperparameter optimization software framework, particularly designed for open-source machine learning models. Optuna enables users to adopt state-of-the-art algorithms for sampling hyperparameters

(g) **Model Monitoring** Model monitoring tools can give us visibility into what is happening in production, including monitoring data and concept drifts and alerts when unexpected behavior happens. Amazon SageMaker Model Monitor allows us to automatically monitor production models and alerts whenever data and model quality issues appear.

(h) **ML Development Best Practices:** Start with a business problem and define a clear evaluation objective. Start with a simple and interpretable model, use simple metrics to meet objectives, use checkpoints for debugging, use Docker and Kubernetes in deployment, and document each feature

4 Experiments to Reproduce Machine Learning Bugs

The software engineering communities have applied ML for software testing, code representation, program comprehension, quality analysis, code clone, and smell detection, code refactoring, and program vulnerability analysis. The source code is a richer construct that contains syntax, structure, and semantics, and therefore should not be treated simply as a collection of tokens or texts [81, 85]. Static code analysis is a selection of algorithms and techniques used to analyze source code without running the program and detect vulnerabilities, errors, or poorly written code at compile time [85]. ML techniques such as model generation, data sampling, feature extraction, model training, and model evaluation are often used for performing source code analysis [81].

4.1 Codebase Analysis of ML GitHub Repositories

We performed codebase analysis of software repositories of ML libraries from GitHub using three popular static code analysis tools: SonarQube [83], DeepSource[33], and CodeScene [4]. Source code analysis with SonarQube, DeepSource, and CodeScene aims to ensure that a software program conforms to high-quality software design principles, such as the open-closed principle, design pattern, and a fewer number of technical debts in order to produce high-quality software with fewer bugs and less expensive to refactor.

(a) SonarCloud: Code Quality and Code Security Platform

SonarCloud is the leading online static analysis tool to catch Security Vulnerabilities, Bugs, and Code Smells in your pull requests, branches, and throughout your repository. It uses quality gates which are standard practices, coding rules, a set of databases for storing analysis reports, and dashboards to report code health. A quality gate is a set of boolean metrics to determine whether a given project is production-ready or not. If the current status of the project is not passed, the set of metrics allows us to see which metric is causing the problem and the values required to pass. The quality gate from the code analysis of the Scikit-Learn repository is given in Figure 26.

(b) **DeepSource: The Modern Static Analysis Platform**

DeepSource is an automated code review tool that helps developers write better code, by finding and automatically fixing bugs and defects before the code is pushed to production [33]. It finds issues such as bug risks, anti-patterns, security vulnerabilities, performance problems, style violations, documentation gaps, and a deficiency of test coverage in source code. The tool uses regular expressions, and ML techniques to detect issues in code, generate fixes and prioritize issues depending on the project(team's) coding conventions.

(c) **CodeScene: A Behavioral Code Analysis Tool** CodeScene automatically identifies and prioritizes critical flaws in software and gives actionable insights for developers to predict risks and improve software quality to align with their key business goals [4]. In contrast to other static code analysis tools, CodeScene is a behavioral code analysis that considers the temporal dimension and evolution of the entire system, rather than a snapshot of the codebase. The tool prioritizes technical debt to ensure that the suggested improvements give you a real return on your investment [4].

We performed a code analysis of Scikit-Learn, Tensorflow, Keras, and PyTorch by establishing a connection to their GitHub repositories, and a few findings are documented in Table 4. SonarCLoud, DeepSource, and CodeScene are all excellent tools that can reveal high-level quality issues in software repositories. The identified bugs are among the most discussed bugs in chapter 2.

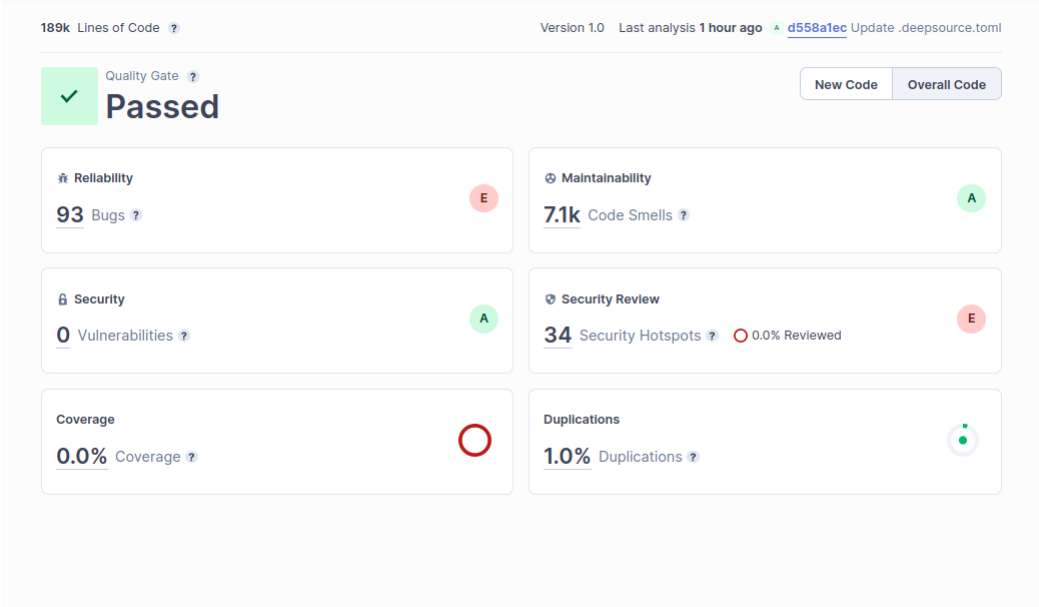

Figure 26: Sonarcloud quality gate for scikit-learn repository

The semantics and definition of selected quality profiles and metrics are:

- A bug is a coding error that can break a code and needs to be fixed immediately. For example, a reliability rating of **E** indicates there is at least one blocker bug.
- A vulnerable code is a code that can be exploited by hackers. For example, a security rating of **D** tells there is at least one critical vulnerability.
- A code smell is a code that is confusing and difficult to maintain. For example, a maintainability rating of **A** is when the technical debt ratio is less than 5.0
- A security hotspot is a sensitive code that requires manual review to assess whether or not a vulnerability exists. For example, a security Review rating of **E** when less than 30 % of Security Hotspots are reviewed.

4.2 Experiments to Reproduce ML Bugs

We replicated papers for metamorphic testing of SVM image classifiers, smoke testing of classification and clustering algorithms, and behavioral testing for NLP models using Jupyter Notebook, PyCharm, and IntelliJ IDEA tools.

(a) Implementation Bug Detection Using Metamorphic Testing

When an ML classifier exhibits an incorrect classification for a given input, there can be multiple underlying reasons for the classification error. Some of the reasons are deficient training data, poor ML architecture used, the ML algorithm learning a wrong function, and there is an implementation bug [36]. An incorrect classification is directly attributed to inadequate training data, and developers are encouraged to collect more diverse training data to mitigate the problem. However, if the incorrect classification that was seen is due to an implementation problem of the algorithms, collecting more training data will not help a lot. Assuring the ML algorithm does not have implementation bugs ensures its correct implementation.

Researchers [36], came up with an approach on how to detect implementation bugs for an image classification model. We replicated on SVM image classification. The application classifies images of handwritten digits into 10 classes. Mutation testing is used to evaluate the fault-revealing ability of a test suite by injecting faulty codes. The ratio of detected faults against all injected faults is

Quality Profiles	SciKit-Learn	Keras	TensorFlow	PyTorch
SonarQube				
Bugs	93	27	228	463
Security Vulnerabilities	30	3	5	1
Code Smells	7.1k	1.5k	9.7k	15K
Security HotSpots	34	42	301	609
Duplication (%)	1.0	6.9	4.4	3.7
Reliability	E	C	E	E
Maintainability	A	A	A	A
Bugs	Argument Error	Missing Argument	Method Call Error	Image Version Tagging
DeepSource				
Bugs	330	1.8k	29	6
Performance Issues	67	60	1	1
Security Issues	1.2k	308	5	3
Design Anti-Patterns	1K	2.3k	28	5
Doc Coverage(%)	16.7	12.2	33.5	40.6
Bug Example	Division by zero	KeyError	Wrong Method Call	Null Pointer Exception
Software Version	1.1.2	2.9.0	2.9.1	1.12

Table 4: ML GitHub repository quality analysis using default branch

called the mutation score [92]. The principle behind mutation testing is that the generated mutants represent the errors programmers typically make to validate the performance and robustness.

The MRs to evaluate the robustness of SVMs are given in Table 5. Ideally, the algorithms trained and evaluated on the original and transformed data should result in the same prediction. The permutation of training instances was sufficient to catch all of the mutants in both linear and non-linear SVMs. This indicates that the mutants correspond to incorrect handling of input data. Permuting the input instances causes the SVM to think the class label has changed, resulting in a different outcome.

In summary, **ValueError** issues are because the number of classes must be greater than 1, but the classifier got one class. The inability of the model to

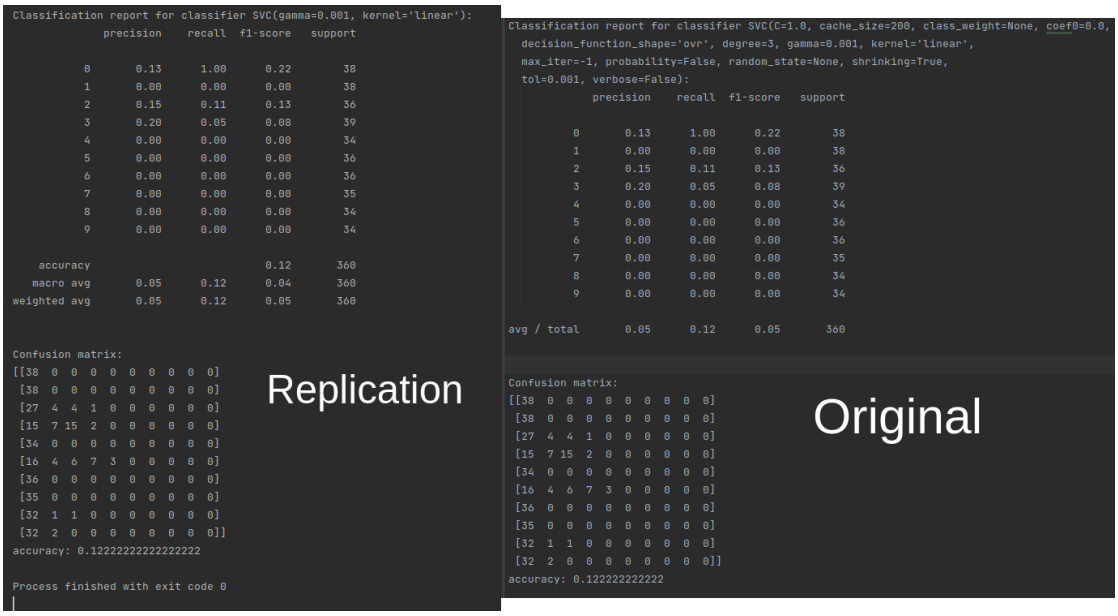

Figure 27: Confusion matrix and classification report

Metamorphic Relations	**Healthy Code**	**Mutant Code**	**Description**
Permute features	digits[:,(-1)]	digits[:,1]	Different column
Permute instances	digits_test[:,(-1)]	digits_test[:,1]	Different column
Shift features	digits[:,(-1)]	digits[:,(-2)]	Different column
scale test features	digits_test[:,(-1)]	digits_test[:,(-2)]	Different Column

Table 5: Metamorphic relations and representative mutant codes

Code	SVM	Avg. Accuracy	Bug	Successful and Non successful Mutants
Normal	Linear	**0.963**	Incompatible shape	-
	Non-Linear	**0.975**	Type conversion	-
Buggy	Linear	**0.118**	ValueError	Permute instances, Shift features Permute features
	Non-Linear	**0.118**	ValueError	Permute instances Feature scaling Permute features

Table 6: Application of metamorphic relations to detect mutant codes

reshape the input array is also another cause. The **incompatible shape** error is due to data transformation incompatibility issues. The **Type conversion** error occurs when the model is unable to handle the data conversion properly.

(b) Evaluating Classification Algorithms With Smoke Testing

To get an in-depth understanding of smoke tests applied to detect bugs that hinder the core functionality of ML algorithms, we replicated **Smoke testing for machine learning: simple tests to discover severe bugs** to check how bugs can be detected in classification and clustering algorithms from Scikit-Learn and Spark ML. The test cases for smoke testing should be designed to cover inputs with values close to machine precision, input features with exactly zero values, features with many categories, etc. A broad set of hyperparameters can be employed to ensure adequate coverage.

The test cases are generated with equivalence class and boundary values analysis to handle the behavior of the algorithms [41]. The equivalence classes were designed in such a way that, four classes deal with features that might trigger problems due to precision issues, three deal with feature distributions that may pose problems for algorithms that infer feature distributions, three classes for categorical features, and two of which deal with problematic labels of classification data independent of feature values [41].

Frameworks	Classification	Clustering	Bugs Impacts
Scikit-Learn (1.1.2)	• Decision Tree • Random Forest • SVM • Ridge • SGD • KNN • Naive Bayes • MLP	• K-Means • DBSCAN • Agglomerative • Gaussian	• Class imbalance • One class problem • Extreme values • All zero values • Memory error • Array out of Index • Attribute Error • Performance
PySpark (3.2.1)	• Decision Trees • Random Forest • GBT • Naive Bayes • Log Regression	• K-Means • B-KMeans • Gaussian	• Unsupported args • Extreme values • One class label • Division by zero • Contains NaN • Convergence • Many categories • Boundary problem • Program crash

Table 7: Experimental results of smoke testing

(c) Testing Sentiment Models for Language Capabilities

NLP models are commonly evaluated on a held-out dataset to determine how well they generalize to unseen data. Assume we reach our target performance metric of 98% on a held-out dataset and deploy the model to production based on this metric. When real users start using it, the story can be different from what the performance metric has indicated. In production, the model may perform poorly even on simple variations of the training data. We replicated [72], a general framework for performing behavioral tests for an NLP model. The tool was applied to Microsoft's Text Analytics, Google Cloud's Natural Language Processing, Roberta, BeRT, and Amazon's Comprehend for sentiment analysis, the Quora Question Pairs, and Machine translations. We experimented with the sentiment analysis task using Jupyter Notebook using the script given below. The code snippet is for running a sentiment analysis with RoBERTa pre-trained model.

```
1 import checklist
2 from checklist.test_suite import TestSuite
```

```
suite_path = 'release_data/sentiment/sentiment_suite.pkl' # Load Suite
suite = TestSuite.from_file(suite_path)
pred_path = 'release_data/sentiment/predictions/roberta' # Select model
suite.run_from_file(pred_path, overwrite=True) # and run Suite
suite.visual_summary_table() # or suite.summary() # Visualize results
```

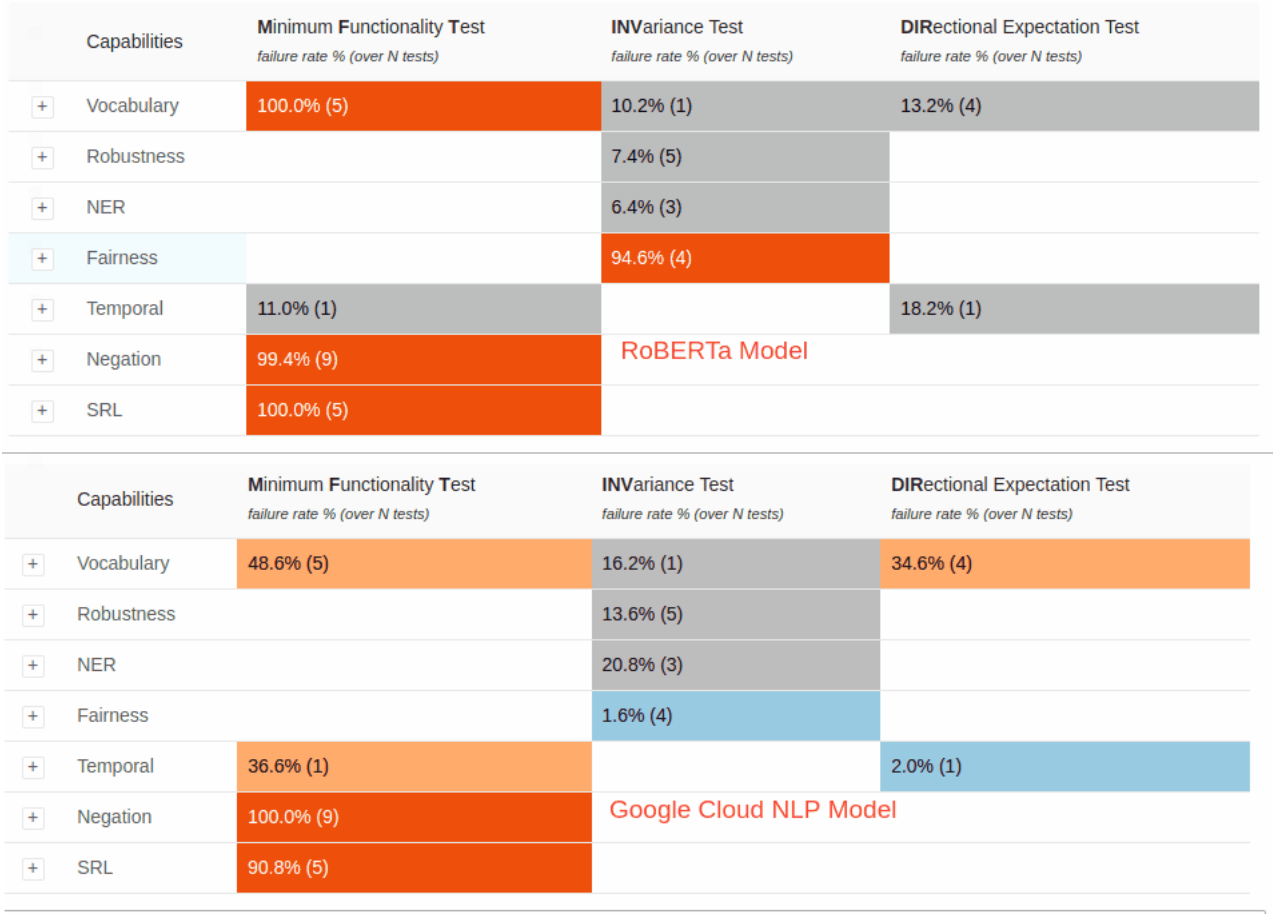

Figure 28: RoBERTa and Google NLP models for sentiment analysis

In Figure 28 and Figure 29 , rows represent model capabilities, while columns represent test types. The intersecting cells contain failure rates from multiple test cases.

From the experimental results shown in Figure 28 and Figure 29, many of the models are unable to correctly predict or understand various language capabilities or constructs in the sentiment analysis tasks.

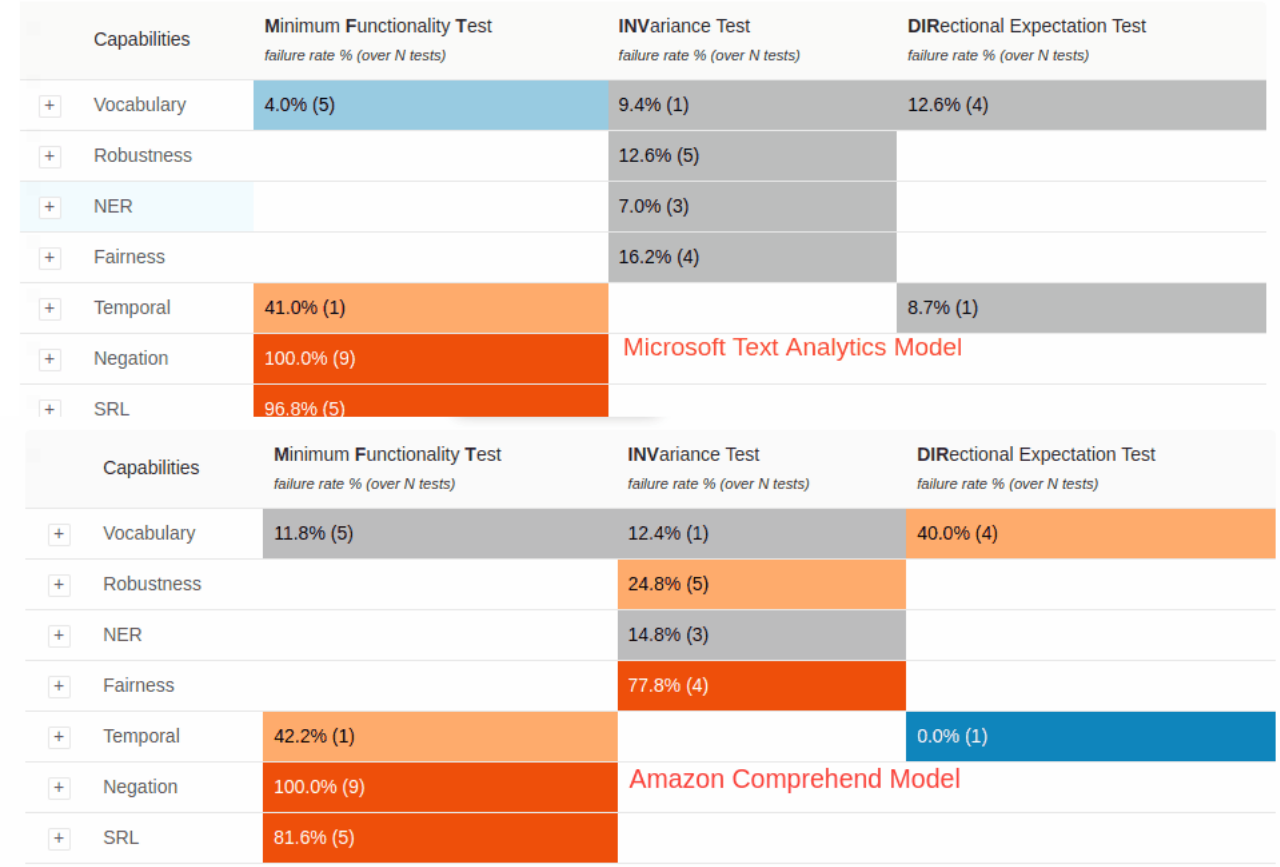

Figure 29: Amazon comprehend and Microsoft text analytics for sentiment analysis

While Checklist was intended for testing NLP models specifically, the concept can be adapted and extended to other models including binary classification models from spark ML, Scikit-Learn, and Keras.

In summary, the three testing methods helped us to understand and examine the implementation, and model prediction bugs of classification algorithms from Scikit-Learn and Spark ML. Moreover, the model behavioral testing helped us to evaluate models beyond model performance metrics such as accuracy.

5 Experimental Design

The purpose of the experimental design is to evaluate 17 different classifiers from Spark ML, Keras, and Scikit-Learn to minimize frequent bugs at the early stages. Model behavioral testing is a method where we test machine-learning software based on the expected inputs and outputs of a model. It is a state-of-the-art testing technique for classification models without worrying about the internal black-box implementations of the algorithms [28, 72, 70].

5.1 Purpose of the Experiments

We took inspiration from the NLP model's behavioral testing [72] and DeepChecks [32] a model and data evaluation tool to evaluate classification algorithms and ML test score [25], a rubric for testing ML systems. The various classification algorithms are evaluated by model pre-train tests to ensure data quality. We also performed model post-train tests to verify if the scikit-learn learned model functions correctly. The purpose of the experiment is to apply model pre-train, post-train tests, and in-depth data validation techniques for 17 ML classification algorithms from open-source ML libraries to build user confidence in using these systems to build and deploy operational ML systems.

We want to answer the following questions. What are the ideal classifiers for the problem at hand, what are the most appropriate evaluation metrics, and what factors make a classifier perform best? We use precision, recall, accuracy, RoC, Confusion matrix, and classification report. What are the methods and parameters that make the black box decision-making process more understandable and interpretable? And, how do the input features contribute to the model output? What are the challenges, and what parameters ensure the reproducibility of models and predictions and have a representative train test split? What are the most appropriate classifiers, ideal performance metrics for the raw data, normalized data, and imbalanced data(class distribution)?

What set of ML properties should ML practitioners consider for getting a holistic view of the model? What is the reason that accuracy alone does not provide a complete picture of a production-level model? Is it possible to tell how each of the

various properties correlates with the classifiers i.e does a decision tree classifier emphasize robustness over explainability?

5.2 Experimental Workflow

The following diagram illustrates the workflow for the prototype implementation

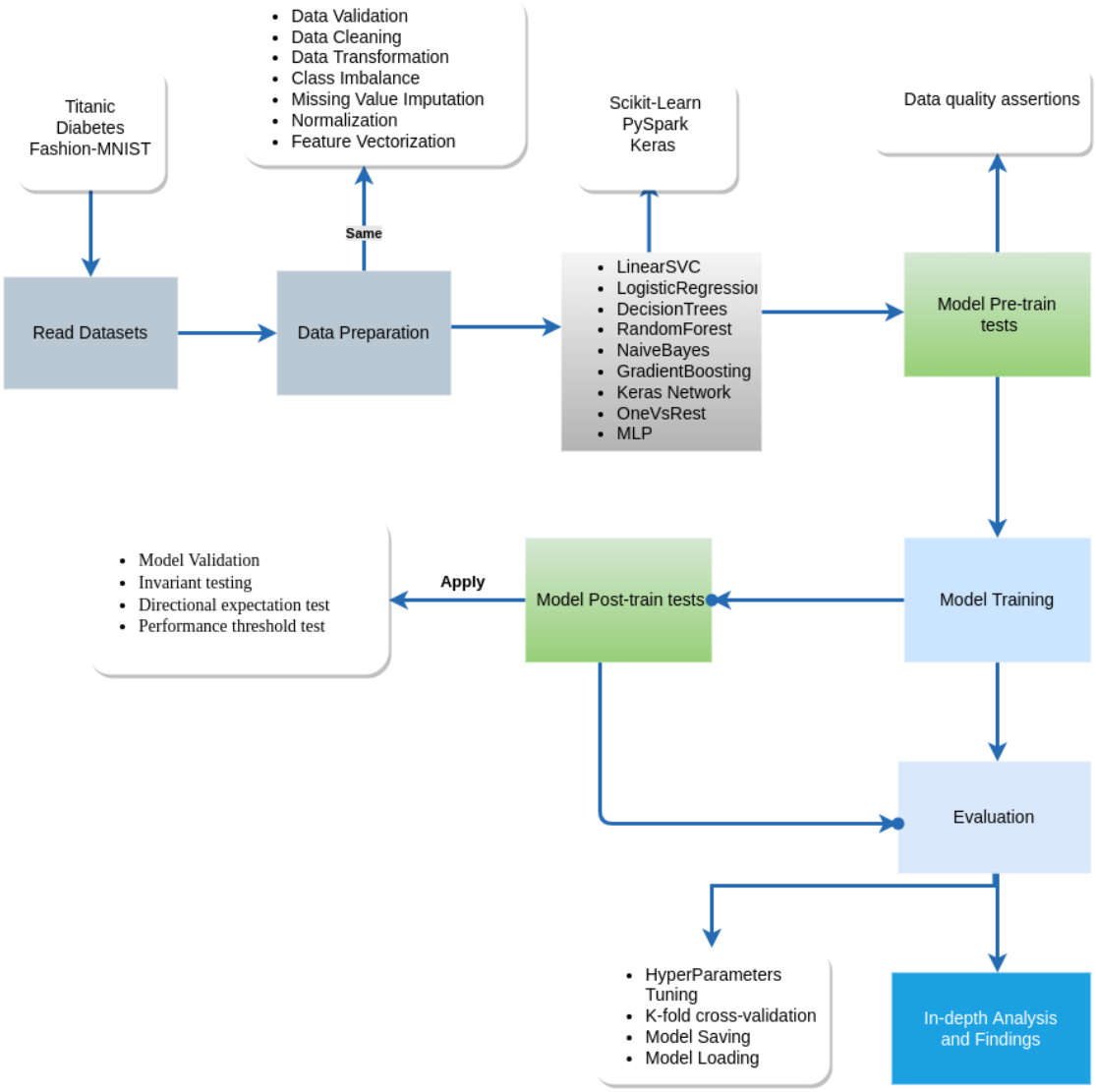

Figure 30: Implementation Workflow Diagram

We selected three popular classification datasets. To avoid errors in the data, we performed systematic data preprocessing and quality checks before feeding the

data to the models. We construct the models and then evaluate them. In parallel, we checked the correctness of the model to check if the model is correctly applying the trained logic by conducting model post-train tests. An in-depth analysis of the various models is the final step in the workflow, using accuracy, recall, precision, and ROC curves to show a holistic view of the various models.

5.2.1 Tools and Software

We employed the following tools to develop the ML prototype.

(a) **Scikit-Learn** is an open-source Python library that implements a range of machine learning algorithms using a unified API.

(b) **PySpark** is a Python API for machine learning applications built on top of Spark ML

(c) **Keras** is an open-source library that provides a Python interface for ANN.

(d) **Python** is a high-level, general-purpose, interpreted, interactive and object-oriented programming language designed to be highly readable.

(e) **Jupyter Notebook** is a browser-based tool for interactive development that combines explanatory text, mathematics, computations, and their rich media output

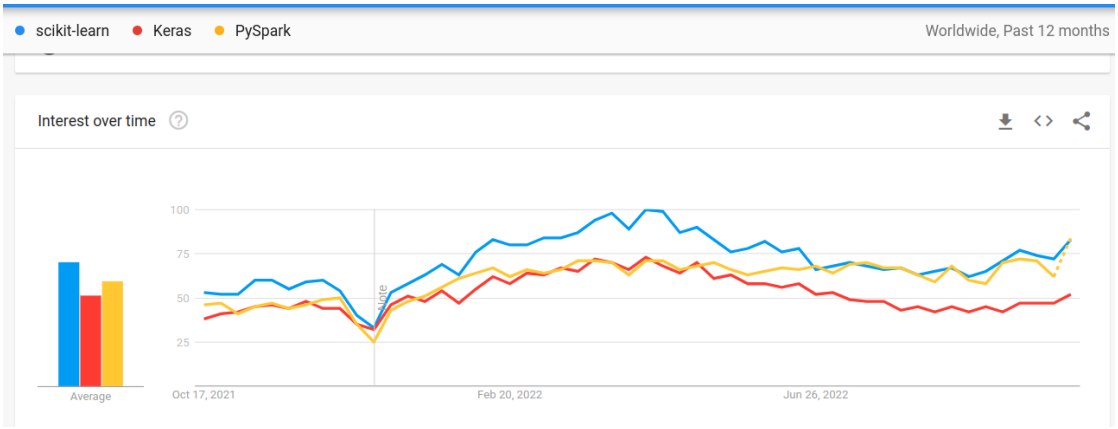

Figure 31: The average user interest over a one-year period

5.2.2 Datasets

The following datasets are used for the experimentation

(a) **Fashion MNIST Dataset** consists of 60,000 training examples and 10,000 test examples of Zalando's article images.

(b) **Titanic Disaster Dataset** describes the survival status of individual passengers as a result of the sinking of the Titanic in early 1912.

(c) **Pima Indian Diabetes Dataset** is used to determine whether or not a patient has diabetes, based on certain diagnostic measurements.

5.2.3 Algorithms

We identified algorithms with similar mathematical interpretation and set of hyperparameters.

Algorithms	PySpark	Scikit-Learn	Description
Logistic Regression	• regParam • maxIter • elasticNetParam	• C • max_iter • penality	• Regularization • Convergence • Penalty norm
Decision Tree Random Forest GBT	• Impurity • maxDepth • numTrees • subsampling Rate • stepsize	• criterion • max_depth • n_estimators • subsample • learning rate	• Information gain • Depth of the trees • Number of tress • Samples • Contribution
LinearSVC	• loss • maxIter • regParam	• penality • max_iter • C	• Penalty norm • Iterations • Regularization
OneVsRest	• estimator • parallelism	• estimator • n_jobs	• Estimator • Parallel jobs
MLP Classifier	• solver • maxIter • activation	• solver • max_iter • activation	• Optimizer • Iterations • Activation
Naive Bayes	• smoothing • model type	• var_smoothing • modeltype	• smoothing • model type
Keras Classifier	• hidden layers • learning rate • activation function • optimizer		

Table 8: Classification Algorithms and their hyperarameters

6 Empirical Experiments

This chapter mainly focuses on the analysis of the experimental results and the rationale behind the outcomes.

6.1 Project Structure

The project implementation can be accessed at ML Systems Quality Evaluation. The arrangement of the project is given below

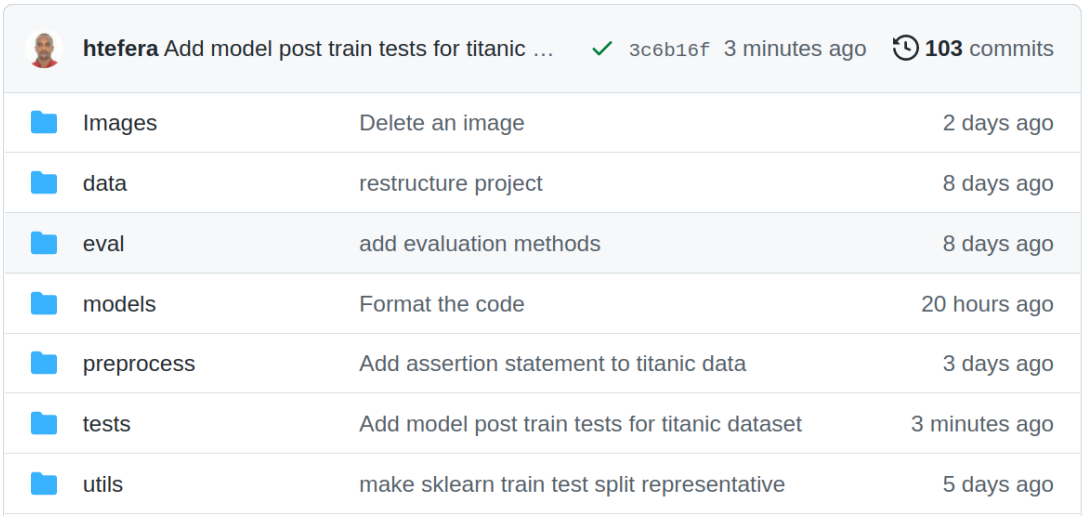

Figure 32: Implementation Project Structure GitHub

(a) **Data** contain the datasets for experimentation

(b) **Eval** Implement metrics and visualizations for evaluation

(c) **Models** Contains implementation logic of classification models

(d) **Preprocess** stores code artifacts for explanatory data analysis

(e) **Tests** model post train test implementation

(f) **Utils** contain data processing logic and some helper functions for the various models and libraries

6.2 Data Preparation and Feature Engineering

We performed the following data processing and feature tasks using Zalando's Fashion-MNIST, Titanic survival status prediction, and Pima Indians Diabetes datasets before we feed them to the various models we build. For report writing, we mainly used the Titanic dataset. First, we have to do the necessary preprocessing to feed the raw data into the Deepchecks to check for data and model drift. For that, we need to remove the null values and change the string datatype columns to numerical. We trained the models without further preprocessing. The following code snippet removes null values and encodes categorical features then we can feed the data to the DeepCheck Datasets suite

```

for i in ["Name", "Sex", "Ticket", "Cabin", "Embarked"]:
    titanic[i] = titanic[i].astype('category').cat.codes
 titanic.head()
```

(a) Deepchecks Reports of Unprocessed Data

Now we can run Deepchecks on unprocessed data using 5 models from scikit-learn and the tool performed 36 different tests. The following code Snippet of data and model validation reports

```

from deepchecks import Dataset
from deepchecks.suites import full_suite

from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(random_state=seed)
clf.fit(x_train, y_train)
dc_train = Dataset(df=x_train, label=y_train, cat_features=[])
dc_test = Dataset(df=x_test, label=y_test, cat_features=[])
fsuite = full_suite()
result = fsuite.run(train_dataset=dc_train, test_dataset=dc_test, model=clf)
result
```

The test results are organized as failed, passed, others, and did not run.

Status	Check	Condition	More Info
✓	Train Test Performance	Train-Test scores relative degradation is less than 0.1	Found max degradation of 3.43% for metric Precision and class 1.
✓	Feature Label Correlation - Test Dataset	Features' Predictive Power Score is less than 0.8	Passed for 11 relevant columns
✓	Feature Label Correlation - Train Dataset	Features' Predictive Power Score is less than 0.8	Passed for 11 relevant columns
✓	Feature-Feature Correlation - Train Dataset	Not more than 0 pairs are correlated above 0.9	All correlations are less than 0.9 except pairs []
✓	Train Test Label Drift	categorical drift score < 0.2 and numerical drift score < 0.1 for label drift	Label's drift score Cramer's V is 0
✓	Train Test Feature Drift	categorical drift score < 0.2 and numerical drift score < 0.1	Passed for 11 columns out of 11 columns. Found column "Name" has the highest numerical drift score: 0.04
✓	Feature Label Correlation Change	Train features' Predictive Power Score is less than 0.7	Passed for 11 relevant columns
✓	Feature Label Correlation Change	Train-Test features' Predictive Power Score difference is less than 0.2	Passed for 11 relevant columns
✓	Category Mismatch Train Test	Ratio of samples with a new category is less or equal to 0%	No relevant columns to check were found
✓	Feature-Feature Correlation - Test Dataset	Not more than 0 pairs are correlated above 0.9	All correlations are less than 0.9 except pairs []
✓	Model Inference Time - Test Dataset	Average model inference time for one sample is less than 0.001	Found average inference time (seconds): 9.34e-06

Figure 33: Successful tests of logistic regression model

There are also a few tests for example train test performance.

Status	Check	Condition	More Info
✗	Train Test Performance	Train-Test scores relative degradation is less than 0.1	1 scores failed. Found max degradation of 10.44% for metric Recall and class 1.
✗	Simple Model Comparison	Model performance gain over simple model is greater than 10%	Found classes with failed metric's gain: {0: {'F1': '4.55%'}}

Status	Check	Condition	More Info
✗	Train Test Performance	Train-Test scores relative degradation is less than 0.1	5 scores failed. Found max degradation of 26.09% for metric Recall and class 1.
✗	Train Test Prediction Drift	categorical drift score < 0.15 and numerical drift score < 0.075	Found model prediction Earth Mover's Distance drift score of 0.09
!	Weak Segments Performance - Test Dataset	The relative performance of weakest segment is greater than 80% of average model performance.	Found a segment with Accuracy score of 0.4 in comparison to an average score of 0.849 in sampled data.

Figure 34: Random forest and LinearSVC failed tests

One of the interesting findings is the existence of train and test data outliers.

	Outlier Probability Score	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived
21	0.94	22	2	80	1	34.00	0	0	152	13.00	111	2	1
699	0.93	700	3	382	1	42.00	0	0	346	7.65	139	2	0
776	0.92	777	3	812	1	-1.00	0	0	481	7.75	143	1	0
429	0.92	430	3	663	1	32.00	0	0	644	8.05	115	2	1
438	0.77	439	1	261	1	64.00	1	4	95	263.00	63	2	0

Figure 35: Outlier samples from Titanic training data

The Outlier Probability Score measures the local deviation of the density of a given sample with respect to its neighbors and the higher the value the most likely the instance is an outlier. These outlier scores are directly interpretable as a probability of an object being an outlier.

We have also seen model overfitting problems for the decision tree and random forest classifiers and Simple model comparison for the **LinearSVC** model. In summary, we used the tools to validate the data and the model which can be used as model post-train tests

(b) Model Pre-train Tests

First, we write a group of assertions to validate the quality of data so that any data given to it could not break the model.

```
assert set(titanic.columns) == set(("PassengerId", "Survived", "Pclass",
    "Name", "Sex", "Age", "SibSp","Parch", "Ticket",
    "Fare", "Cabin", "Embarked")), "Unexpected Column(s)"
assert titanic[["PassengerId", "Survived", "Pclass", "Name",
"Sex", "SibSp", "Parch", "Ticket", "Fare"]].isna().sum().sum() == 0,\
    "Missing values should only be in columns: Age, Cabin and Embarked"
assert set(titanic["Sex"].unique()) == set(
    ("female", "male")), "Unknown gender identified"
assert set(titanic["Survived"].unique()) == set(
    (0, 1)), "Unknown survived value recieved"
assert set(titanic["Embarked"].unique()) == set(
    ("S", "C", "Q", np.nan)), "Unknown mbarked location"
assert set(titanic["Pclass"].unique()) == set(
    (1, 2, 3)), "Unknown Pclass value"
assert titanic["Age"].min() >= 0, "Age should be positive"
assert titanic["SibSp"].min() >= 0, "SibSp should be positive"
assert titanic["Fare"].min() >= 0, "Fare should be positive"
assert titanic["Parch"].min() >= 0, "Parch should be positive"
assert titanic["Parch"].min() <= 6, "Parch should be less than 7"
```

(c) Data Preparation and Feature Engineering

We have to apply proper data cleaning and preprocessing techniques and apply feature engineering to get meaningful data out of raw data. There are some missing values in the age, cabin, and embarked columns. We dropped the cabin

column. Instead of taking the general mean, we use the names column to see the title of the person and assign the mean age according to that.

```
titanic["Age"][titanic["Name"].str.contains("Mr.")
               & ~titanic["Sex"].str.contains("female")
               & titanic["Age"].isna()] = round(titanic["Age"][
                   titanic["Name"].str.contains("Mr.")
                   & ~titanic["Sex"].str.contains("female")].mean())
titanic["Age"][titanic["Name"].str.contains("Master.")
               & ~titanic["Sex"].str.contains("female")
               & titanic["Age"].isna()] = round(titanic["Age"][
                   titanic["Name"].str.contains("Master.")
                   & ~titanic["Sex"].str.contains("female")].mean())
titanic["Age"][titanic["Name"].str.contains("Mrs.")
               & titanic["Sex"].str.contains("female")
               & titanic["Age"].isna()] = round(titanic["Age"][
                   titanic["Name"].str.contains("Mrs.")
                   & titanic["Sex"].str.contains("female")].mean())
titanic["Age"][titanic["Name"].str.contains("Miss.")
               & titanic["Sex"].str.contains("female")
               & titanic["Age"].isna()] = round(titanic["Age"][
                   titanic["Name"].str.contains("Miss.")
                   & titanic["Sex"].str.contains("female")].mean())
titanic["Age"][~titanic["Name"].str.contains("Miss.")
               & ~titanic["Name"].str.contains("Mr.")
               & ~titanic["Name"].str.contains("Master.")
               & titanic["Age"].isna()] = round(titanic["Age"][
                   ~titanic["Name"].str.contains("Miss.")
                   & ~titanic["Name"].str.contains("Mr.")
                   & ~titanic["Name"].str.contains("Master.")].mean())
```

Since **Embarked** is a categorical column and there are only two missing values, so, instead of looking for the mean value, we have to look for the most repeated value(mode) to input the missing value. The passenger Id and name columns have **891** values which are exactly equal to the number of rows. This means that every row contains a unique value and we remove them.

We check for the rows that are entirely the same as some others to avoid data leakage from the train to the test dataset. To make sure if there are any two

people who have the same set of conditions but one of them survived and the other did not.

```
1 titanic[titanic.drop(columns="Survived").duplicated()]
```

	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked
324	0	3	male	32.00	8	2	CA. 2343	69.5500	S
409	0	3	female	22.00	3	1	4133	25.4667	S
413	0	2	male	32.00	0	0	239853	0.0000	S
466	0	2	male	32.00	0	0	239853	0.0000	S
485	0	3	female	22.00	3	1	4133	25.4667	S
612	1	3	female	22.00	1	0	367230	15.5000	Q
641	1	1	female	24.00	0	0	PC 17477	69.3000	C
643	1	3	male	32.00	0	0	1601	56.4958	S
644	1	3	female	0.75	2	1	2666	19.2583	C
692	1	3	male	32.00	0	0	1601	56.4958	S
709	1	3	male	5.00	1	1	2661	15.2458	C
792	0	3	female	22.00	8	2	CA. 2343	69.5500	S
826	0	3	male	32.00	0	0	1601	56.4958	S
838	1	3	male	32.00	0	0	1601	56.4958	S
846	0	3	male	32.00	8	2	CA. 2343	69.5500	S
863	0	3	female	22.00	8	2	CA. 2343	69.5500	S

Figure 36: Duplicate values detection

There is a male at index **826** who shares all the conditions with a survivor at index **838** but couldn't survive himself. These contradictions can confuse the models, so it is better to remove one of them.

```
1 titanic.head()
```

:

	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked
0	0	3	male	22.0	1	0	A/5 21171	7.2500	S
1	1	1	female	38.0	1	0	PC 17599	71.2833	C
2	1	3	female	26.0	0	0	STON/O2. 3101282	7.9250	S
3	1	1	female	35.0	1	0	113803	53.1000	S
4	0	3	male	35.0	0	0	373450	8.0500	S

Figure 37: The first five rows of the dataset

Now that all the duplicated entries are removed as well, we can work on converting the string columns to integer or float because the ML models better understand numerical values. There are 3 object columns that contain string values: sex, ticket and embarked. We can do the encoding to convert the columns to integer values.

```
1 titanic["Sex"] = np.where(titanic["Sex"] == "male", 1, 0)
2 titanic["Ticket"].replace(list(titanic["Ticket"].value_counts().index),
3                           range(len(titanic["Ticket"].value_counts())),
4                           inplace=True)
5 titanic["Embarked"].replace(list(titanic["Embarked"].value_counts().index),
6                             range(len(titanic["Embarked"].value_counts())),
7                             inplace=True)
```

```
1 titanic.head()
```

	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked
0	0	3	0	22.0	1	0	237	7.2500	0
1	1	1	0	38.0	1	0	540	71.2833	1
2	1	3	0	26.0	0	0	487	7.9250	0
3	1	1	0	35.0	1	0	126	53.1000	0
4	0	3	0	35.0	0	0	488	8.0500	0

Figure 38: The first five rows of the encoded dataset

Now that the data is all numeric and from the statistical report using **titanic.describe()** followed encoding, most of the columns have a maximum greater than 1 and the mean value is also very high. So we can normalize the data to remain between 0 and 1 and then apply standardization to make the mean value equal to 0.

```
col = list(titanic.columns)
col.remove("Survived")
for i in col:
    titanic[i] = (titanic[i] - titanic[i].mean()) / titanic[i].std()
```

Now that all the features have a mean of zero and a standard deviation of one, let us check the data after standardization

	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked
0	0	0.834512	0.734762	-0.587123	0.525587	-0.466208	-0.179347	-0.495383	-0.569128
1	1	-1.551756	-1.359429	0.615572	0.525587	-0.466208	1.245180	0.785120	1.001809
2	1	0.834512	-1.359429	-0.286449	-0.494122	-0.466208	0.996006	-0.481885	-0.569128
3	1	-1.551756	-1.359429	0.390066	0.525587	-0.466208	-0.701203	0.421500	-0.569128
4	0	0.834512	0.734762	0.390066	-0.494122	-0.466208	1.000707	-0.479385	-0.569128
...	...	...	...	...	...	...	...	...	...
886	0	-0.358622	0.734762	-0.211281	-0.494122	-0.466208	0.083932	-0.380398	-0.569128
887	1	-1.551756	-1.359429	-0.812628	-0.494122	-0.466208	0.088634	-0.040441	-0.569128
888	0	0.834512	-1.359429	-0.587123	0.525587	2.021180	-0.992690	-0.171425	-0.569128
889	1	-1.551756	0.734762	-0.286449	-0.494122	-0.466208	0.093335	-0.040441	1.001809
890	0	0.834512	0.734762	0.164561	-0.494122	-0.466208	1.903378	-0.485385	2.572746

875 rows × 9 columns

Figure 39: Data after standardization

Now we can look for all the outliers, which can affect the classification models badly. Since the data set is really small, we only aim to remove the 1 percentile outliers on both extremes from each column.

```
temp = titanic
for i in titanic.columns:
    temp = titanic[(titanic[i]<titanic[i].quantile(0.99))
                   & (titanic[i]>titanic[i].quantile(0.01))]
temp
```

As shown in the Figure 40, it seems like most of the values are removed and we are only left with 165 rows, out of 875.

	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked
1	1	-1.551756	-1.359429	0.615572	0.525587	-0.466208	1.245180	0.785120	1.001809
9	1	-0.358622	-1.359429	-1.188470	0.525587	-0.466208	-0.691800	-0.039025	1.001809
19	1	0.834512	-1.359429	0.465235	-0.494122	-0.466208	1.043020	-0.495883	1.001809
26	0	0.834512	0.734762	0.164561	-0.494122	-0.466208	1.028915	-0.495883	1.001809
30	0	-1.551756	0.734762	0.765908	-0.494122	-0.466208	0.977200	-0.086019	1.001809
...	...	...	...	...	...	...	...	...	...
866	1	-0.358622	-1.359429	-0.211281	0.525587	-0.466208	0.022814	-0.363234	1.001809
874	1	-0.358622	-1.359429	-0.136112	0.525587	-0.466208	-0.917468	-0.160426	1.001809
875	1	0.834512	-1.359429	-1.113302	-0.494122	-0.466208	0.051022	-0.495883	1.001809
879	1	-1.551756	-1.359429	1.968603	-0.494122	0.777486	-0.908065	1.022590	1.001809
889	1	-1.551756	0.734762	-0.286449	-0.494122	-0.466208	0.093335	-0.040441	1.001809

165 rows × 9 columns

Figure 40: Data after 1 percent of outliers on both extremes are removed

Instead of looking for outliers in each column, we can look at all columns collectively and then find the outliers. Thus, we can sum all the entries row-wise to get a single value and then find the outliers and remove the 10 percent.

```
add = titanic.drop(columns="Survived").sum(axis=1)
titanic = titanic[(add < add.quantile(0.9)) & (add > add.quantile(0.1))]
```

Now, even after removing the ten percentile of the extreme outliers, we are still left with about 700 rows as shown in Figure 41.

	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Embarked
0	0	0.834512	0.734762	-0.587123	0.525587	-0.466208	-0.179347	-0.495383	-0.569128
1	1	-1.551756	-1.359429	0.615572	0.525587	-0.466208	1.245180	0.785120	1.001809
2	1	0.834512	-1.359429	-0.286449	-0.494122	-0.466208	0.996006	-0.481885	-0.569128
4	0	0.834512	0.734762	0.390066	-0.494122	-0.466208	1.000707	-0.479385	-0.569128
6	0	-1.551756	0.734762	1.818266	-0.494122	-0.466208	1.010110	0.396753	-0.569128
...	...	...	...	...	...	...	...	...	...
883	0	-0.358622	0.734762	-0.136112	-0.494122	-0.466208	0.074529	-0.430392	-0.569128
884	0	0.834512	0.734762	-0.361618	-0.494122	-0.466208	0.079231	-0.499383	-0.569128
886	0	-0.358622	0.734762	-0.211281	-0.494122	-0.466208	0.083932	-0.380398	-0.569128
888	0	0.834512	-1.359429	-0.587123	0.525587	2.021180	-0.992690	-0.171425	-0.569128
889	1	-1.551756	0.734762	-0.286449	-0.494122	-0.466208	0.093335	-0.040441	1.001809

699 rows × 9 columns

Figure 41: Data after ten percent extremes outliers removed

Now it is time to check for the imbalance of data in the output labels, i.e the survived class. To do that we can use a count plot or plot a histogram.

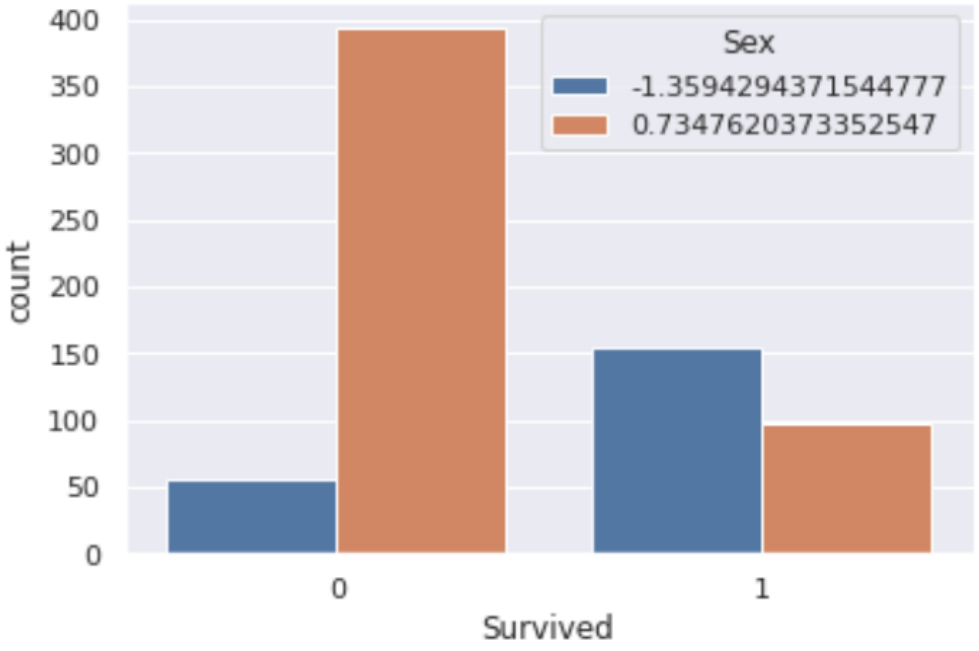

Figure 42: Countplot of processed data

It seems that there is a huge imbalance between the survivors and the ones

that did not. This will create a bias in our models. So we need to balance it by taking fewer samples of the ones that didn't survive. And the sex feature greatly impacts the prediction of the model. That means if the person is a woman then there is a higher probability that she survived, while most men didn't. So, we need to balance the data by taking fewer samples of the ones that did not survive. In machine learning, data resampling by taking fewer samples from the minority class is called **downsampling**

```
titanic = pd.concat([titanic[titanic['Survived'] == 0]
    .sample(len(titanic[titanic['Survived'] == 1]), \
random_state=seed), titanic[titanic['Survived'] == 1]], axis=0)
titanic["Survived"].value_counts()
```

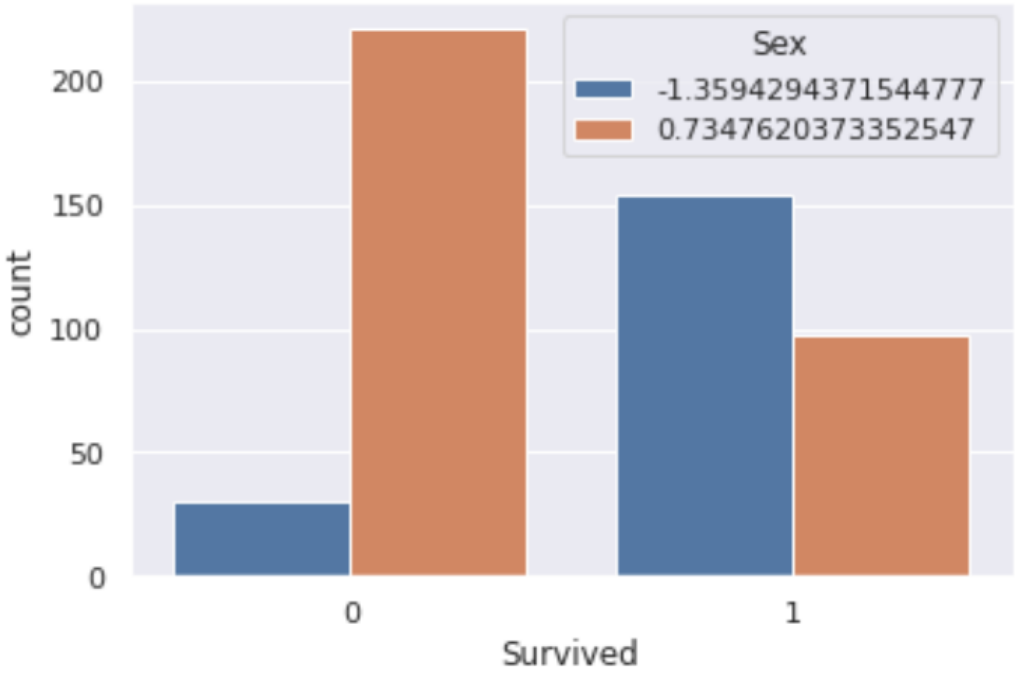

Figure 43: Balanced labels

Now both the labels are balanced with 251 samples each.

We performed similar operations for the three datasets with some slight differences based on the nature of the datasets.

6.3 Training and Evaluation

We build classification models for 17 classifiers from Scikit-learn, Keras and PySpark using three datasets. For convenience purposes, we use the 17 models from the Titanic datasets only.

(a) Import Datasets

We have built two functions saved in **dataset.py** file which preprocesses the data for both sklearn and pyspark and loads it into train and test data accordingly.

```
x_train, x_test, y_train, y_test = datasets.get_sklearn_titanic()
train, test = datasets.get_pyspark_titanic(sc)
```

We used Pandas DataFrames to perform data preprocessing for pyspark and sklearn. In addition, library-specific processing techniques were also employed such as categorical column encoding and vectorization. In PySpark we used **VectorAssembler**, a feature transformer that merges multiple columns into a vector column.

(b) Define Baseline Models

After importing the libraries and the preprocessed titanic dataset we can import both the sklearn and pyspark models and initialize them with only the required hyperparameters, and leave the rest as default to get baseline accuracy. But we have set the random_state or seed value to ensure reproducibility.

```
sklearn_classifiers = [
    sklearn_models.LinearSVC(random_state=seed),
    sklearn_models.LogisticRegression(random_state=seed),
    sklearn_models.DecisionTreeClassifier(random_state=seed),
    sklearn_models.RandomForestClassifier(random_state=seed),
    sklearn_models.GaussianNB(),
    sklearn_models.GradientBoostingClassifier(random_state=seed),
    sklearn_models.MLPClassifier(random_state=seed),
    sklearn_models.OneVsRestClassifier(sklearn_models.LinearSVC())]
pyspark_classifiers = [
    pyspark_models.LinearSVC(labelCol="label"),
    pyspark_models.LogisticRegression(labelCol="label"),
    pyspark_models.DecisionTreeClassifier(labelCol="label", seed=seed),
```

```
    pyspark_models.RandomForestClassifier(labelCol='label', seed=seed),
    pyspark_models.NaiveBayes(labelCol='label', modelType="gaussian"),
    pyspark_models.GBTClassifier(labelCol='label', seed=seed),
    pyspark_models.MultilayerPerceptronClassifier(
        labelCol='label',seed=seed,
        layers=[
            len(test.toPandas()["features"][0]),
            100,]),
    pyspark_models.OneVsRest(labelCol='label',
                             classifier=pyspark_models.LinearSVC())]
```

sklearn_models and **pyspark_models** are accessory methods we defined.

(c) Model Training and Evaluation

We define, train, evaluate and save the accuracy, confusion matrices, RoC, precision, and recall using the test data. We have built a file called **eval_methods.py** to evaluate and visualize the performance metrics.

```
sklearn_accuracy = []
sklearn_confusion = []
sklearn_roc = []
sklearn_precision = []
sklearn_recall = []
_, ax = plt.subplots(figsize=(20, 10))
_, bx = plt.subplots(figsize=(20, 10))
for clf in sklearn_classifiers:
    clf.fit(x_train, y_train)
    accuracy, confusion, roc, precision, recall = eval_methods.eval(
        clf, y_test, clf.predict(x_test), ax, bx)
    sklearn_accuracy.append(accuracy)
    sklearn_confusion.append(confusion)
    sklearn_roc.append(roc)
    sklearn_precision.append(precision)
    sklearn_recall.append(recall)

clf = keras.Sequential()
clf.add(keras.layers.Dense(1024, activation='relu'))
clf.add(keras.layers.Dense(512, activation='relu'))
clf.add(keras.layers.Dense(128, activation='relu'))
clf.add(keras.layers.Dense(32, activation='relu'))
```

```
clf.add(keras.layers.Dense(1, activation='sigmoid'))

clf.compile(optimizer=keras.optimizers.Adam(),
            loss=keras.losses.BinaryCrossentropy(),
            metrics=keras.metrics.BinaryAccuracy())

clf.fit(x_train, y_train, verbose=0)
accuracy, confusion, roc, precision, recall = eval_methods.eval(
    clf, y_test,
    np.round(clf.predict(x_test, verbose=0))[:, 0], ax, bx)
sklearn_accuracy.append(accuracy)
sklearn_confusion.append(confusion)
sklearn_roc.append(roc)
sklearn_precision.append(precision)
sklearn_recall.append(recall)
```

Similarly, we train, evaluate, save the performance metrics and and visualize the pyspark models

```

pyspark_accuracy = []
pyspark_confusion = []
pyspark_roc = []
pyspark_precision = []
pyspark_recall = []
_, ax = plt.subplots(figsize=(20, 10))
_, bx = plt.subplots(figsize=(20, 10))
for clf in pyspark_classifiers:
    clf = clf.fit(train)
    real = np.array([
        1 if "1" in str(x) else 0
        for x in clf.transform(test).select("label").collect()])
    pred = np.array([
        1 if "1" in str(x) else 0
        for x in clf.transform(test).select("prediction").collect()])
    accuracy, confusion, roc, precision, recall = eval_methods.eval(
        clf, real, pred, ax, bx)
    pyspark_accuracy.append(accuracy)
    pyspark_confusion.append(confusion)
    pyspark_roc.append(roc)
    pyspark_precision.append(precision)
```

```
    pyspark_recall.append(recall)

clf = keras.Sequential()
clf.add(keras.layers.Dense(1024, activation='relu'))
clf.add(keras.layers.Dense(512, activation='relu'))
clf.add(keras.layers.Dense(128, activation='relu'))
clf.add(keras.layers.Dense(32, activation='relu'))
clf.add(keras.layers.Dense(1, activation='sigmoid'))

clf.compile(optimizer=keras.optimizers.Adam(),
            loss=keras.losses.BinaryCrossentropy(),
            metrics=keras.metrics.BinaryAccuracy())

clf.fit(x_train, y_train, verbose=0)
accuracy, confusion, roc, precision, recall = eval_methods.eval(
    clf, y_test,
    np.round(clf.predict(x_test, verbose=0))[:, 0], ax, bx)
pyspark_accuracy.append(accuracy)
pyspark_confusion.append(confusion)
pyspark_roc.append(roc)
pyspark_precision.append(precision)
pyspark_recall.append(recall)
```

We have used Keras in both libraries to improve average accuracy, otherwise, the performance was poor and the accuracy was varying a lot.

(d) Baseline Models Result Interpretation

Let us visualize the Area Under the Curve values. The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between the classes and is used as a summary of the ROC curve. For the Sklearn models, the decision tree model shows some outliers at the beginning of the curve. The gradient-boosted tree classifier and the Keras network from pyspark models are less performant when it comes to identifying the two classes. Gaussian Naive Bayes and gradient boosting models from SkLearn classify the survived and not survived classes well, followed by MLP and logistic regression from PySpark and Keras networks from SkLearn.

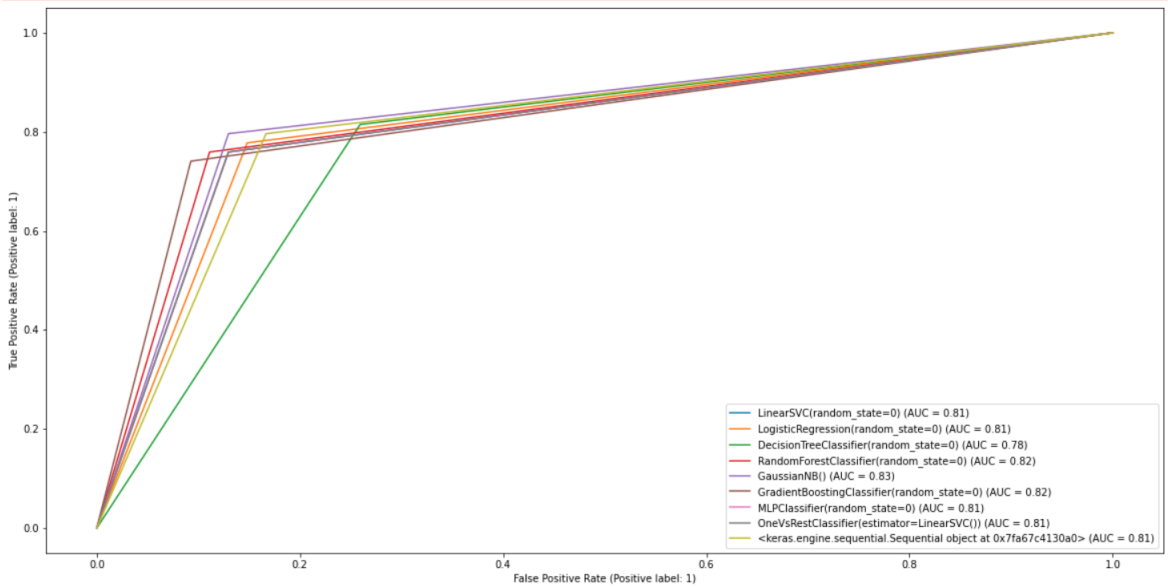

Figure 44: RoC Curve of the sklearn models

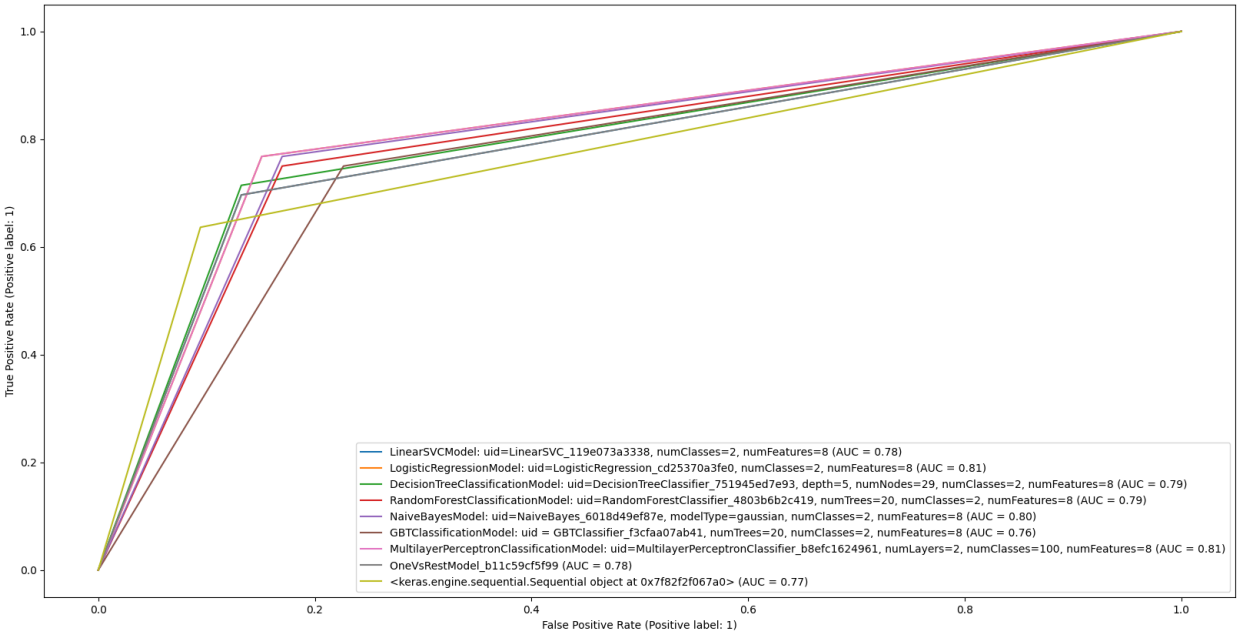

Figure 45: RoC Curve of the pyspark models

Now we can compare the baseline models using the accuracy metrics

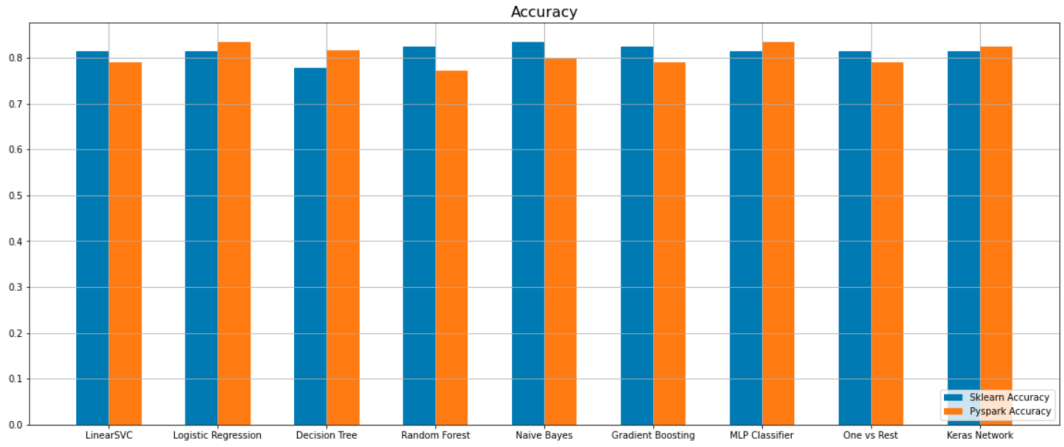

Figure 46: Accuracy of baseline models

The logistic regression, decision trees, Random forest, and, Keras network, MLP models from pyspark slightly outperform their corresponding models in sklearn. In contrast, the Naive Bayes, random forest, OneVsrest, and gradient boosting classifiers from sklearn slightly outperform their corresponding models in pyspark. Finally, the Keras network from sklearn outperforms the corresponding model in pyspark. In general, the performance of models from pyspark and sklearn are comparable.

There is also a similar pattern of the models when examining the RoC report, precision, and recall values.

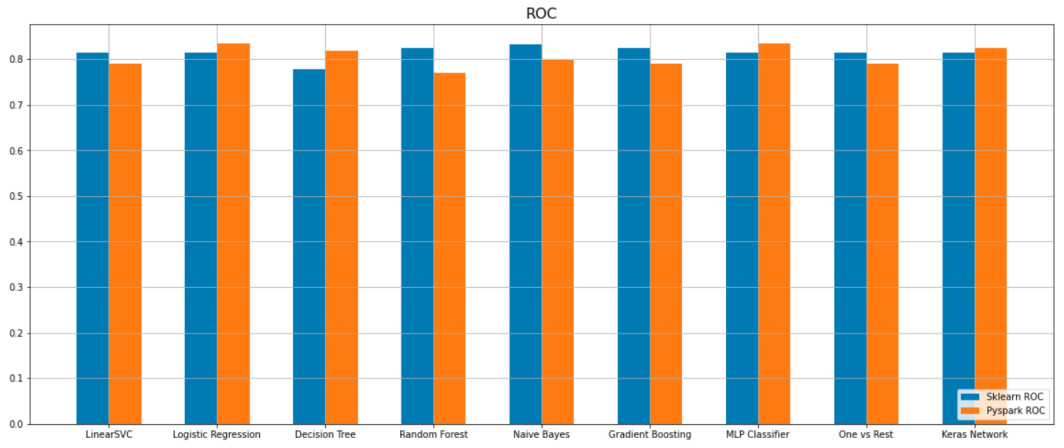

Figure 47: ROC of the baseline models

6.4 Hyperparameter Tuning and Cross Validation

(a) Define and Match Hyperparameters

Now we can define some hyperparameters and their values for the GridsearchCV algorithm. These selected hyperparameters are common to both APIs, i.e. PySpark and Sklearn. The grid search is performed on sklearn and the same combination of hyperparameters is applied to pyspark models to keep the comparison fair.

Before we apply the hyperparameters to the PySpark algorithms, we need to find a match first using the key-value pairs defined in the **para** variable. This is due to there are differences in the name of algorithms as well as the hyperparameters from scikit-learn and PySpark. Therefore, to avoid such confusion we need to find a match of hyperparameters defined for scikit-learn before we apply model tuning for the PySpark algorithms. First, we define common hyperparameters first from sklean.

```
grid = {
    "LinearSVC": {
        "max_iter": [10, 50, 100, 500],
        "tol": [1e-2, 1e-3, 1e-4, 1e-5],
        "C": [0.1, 0.01, 0.001, 0.0001],
        "fit_intercept": [True, False]},
    "LogisticRegression": {
        "max_iter": [10, 50, 100, 500],
        "tol": [1e-2, 1e-3, 1e-4, 1e-5],
        "C": [0.1, 0.01, 0.001, 0.0001],
        "fit_intercept": [True, False]},
    "DecisionTreeClassifier": {
        "max_depth": [2, 5, 10, 20],
        "criterion": ["gini", "entropy"]},
    "RandomForestClassifier": {
        "bootstrap": [True, False],
        "max_depth": [2, 5, 10, 20],
        "criterion": ["gini", "entropy"],
        "n_estimators": [5, 10, 50, 100]},
    "GaussianNB": {
        "var_smoothing": [1e-07, 1e-08, 1e-09, 1e-10]},
    "GradientBoostingClassifier": {
        "max_features": ["auto", "sqrt", "log2"],
```

```
        "max_depth": [2, 5, 10, 20],
        "learning_rate": [0.1, 0.01, 0.001, 0.0001]},
    "MLPClassifier": {
        "tol": [1e-2, 1e-3, 1e-4, 1e-5],
        "max_iter": [10, 50, 100, 500],
        "batch_size": [8, 32, 128, 512]},
    "OneVsRestClassifier": {
        "n_jobs": [1, 3, 5, 7, 10]}}

```

Pyspark hyperparameters are defined next, and their names are matched to those in scikit-learn.

```

para = {
    "LinearSVC": {
        "max_iter": ["maxIter"],
        "tol": ["tol"],
        "C": ["regParam"],
        "fit_intercept": ["fitIntercept"]},
    "LogisticRegression": {
        "max_iter": ["maxIter"],
        "tol": ["tol"],
        "C": ["regParam"],
        "fit_intercept": ["fitIntercept"]},
    "DecisionTreeClassifier": {
        "max_depth": ["maxDepth"],
        "criterion": ["impurity"]},
    "RandomForestClassifier": {
        "bootstrap": ["bootstrap"],
        "max_depth": ["maxDepth"],
        "criterion": ["impurity"],
        "n_estimators": ["numTrees"]},
    "GaussianNB": {
        "var_smoothing": ["smoothing"]},
    "GradientBoostingClassifier": {
        "max_features": ["featureSubsetStrategy"],
        "max_depth": ["maxDepth"],
        "learning_rate": ["stepSize"]},
    "MLPClassifier": {
        "tol": ["tol"],
```

```
        "max_iter": ["maxIter"],
        "batch_size": ["blockSize"]},
    "OneVsRestClassifier": {
        "n_jobs": ["parallelism"]}}

```

Now, we need to find a match between the hyperparameters and algorithms from the key-value pair defined in the **para** dictionary to perform hyperparameter tuning for the pyspark models.

```
grid = {}
for i in para.keys():
    if i == "GaussianNB":
        grid["NaiveBayes"] = {}
        for j in para[i].values():
            grid["NaiveBayes"][j[0]] = j[1]
    elif i == "GradientBoostingClassifier":
        grid["GBTClassifier"] = {}
        for j in para[i].values():
            grid["GBTClassifier"][j[0]] = j[1]
    elif i == "MLPClassifier":
        grid["MultilayerPerceptronClassifier"] = {}
        for j in para[i].values():
            grid["MultilayerPerceptronClassifier"][j[0]] = j[1]
    elif i == "OneVsRestClassifier":
        grid["OneVsRest"] = {}
        for j in para[i].values():
            grid["OneVsRest"][j[0]] = j[1]
    else:
        grid[i] = {}
        for j in para[i].values():
            grid[i][j[0]] = j[1]

```

(b) Train, Evaluate and Save Optimized Models

We apply grid search from the GridSearchCV API to sklearn algorithms, with 5-fold cross-validation (CV). The k-fold CV is used to train the models on the k-1 folds and evaluate the models on the 1 fold of the validation set, where accuracy, precision, recall, and ROC are calculated by taking the average over the 5 folds. And then the models are trained using the best combination of

hyperparameters. The performance metrics and models are saved as .sav files with the Pickle object.

```
for clf in sklearn_classifiers:
    rf_cv = GridSearchCV(estimator=clf,
    param_grid=grid[str(clf)[:str(clf).find("(")]], cv= 5)
    rf_cv.fit(x_train, y_train)
    clf = clf.set_params(**rf_cv.best_params_)
    clf.fit(x_train, y_train)
    path = "./models/titanic/sklearn/"
    if not os.path.exists(path):
        os.makedirs(path)
    pickle.dump(clf, open(path+str(clf)[:str(clf).find("(")]+".sav", 'wb'))
    for i, v in rf_cv.best_params_.items():
        para[str(clf)[:str(clf).find("(")]][i].append(v)
    print(str(clf)[:str(clf).find("(")], rf_cv.best_params_)
    accuracy, confusion, roc, precision, recall =
    eval_methods.eval(clf, y_test, clf.predict(x_test), ax, bx)
    sklearn_accuracy.append(accuracy)
    sklearn_confusion.append(confusion)
    sklearn_roc.append(roc)
    sklearn_precision.append(precision)
    sklearn_recall.append(recall)
```

We performed the same for the pyspark models and we found similar optimal hyperparameter combinations such as max_depth of the decision tree classifier is 2 in both libraries. Now we want to examine the accuracy of the optimized models.

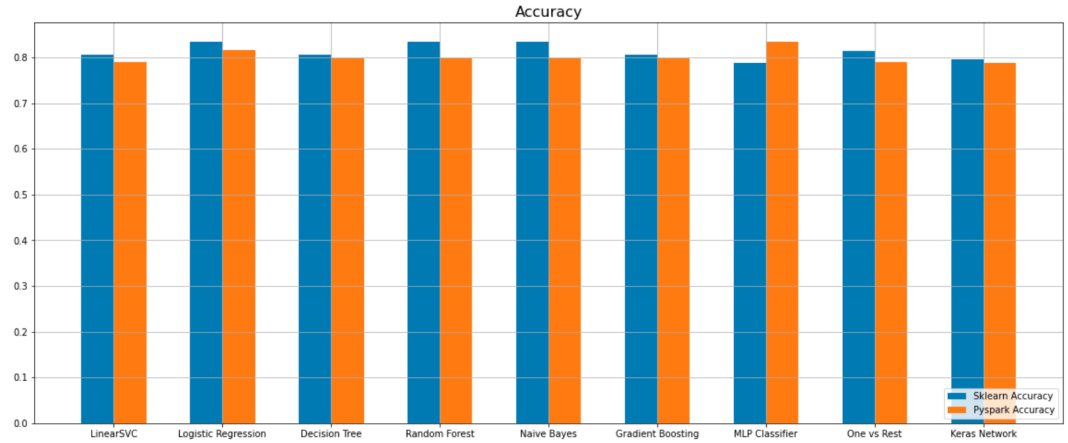

Figure 48: Accuracy graph of optimized models

From the optimized models, the LinearSVC, Logistic regression, Random Forest, Naive Bayes, and One vs Rest models from sklearn outperform their corresponding models in pyspark. In contrast, the MLP model from PySpark outperforms the corresponding model in Sklearn. Finally, the Keras network, the decision tree, and the gradient boosting models are comparable. In summary, the performance of models from pyspark and sklearn are comparable, however, the hyperparameter tuning and k-fold cross-validation technique benefit the models from sklearn. This may be due to the PySpark API applying internal hyperparameter tuning even for baseline models.

The above pattern was repeated for recall, precision, and ROC reports. Let us examine the recall graph because we want to examine how the models spot real survivors.

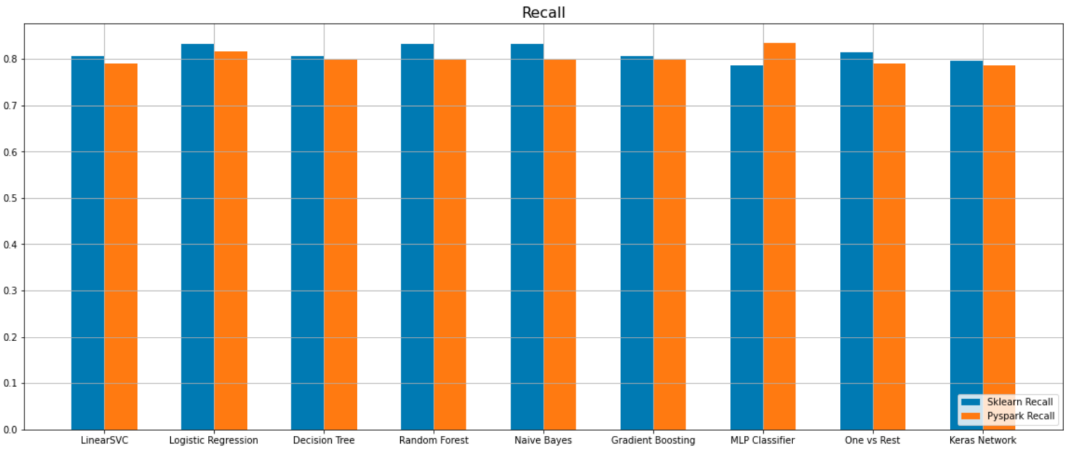

Figure 49: Recall graph of optimized models

6.5 Model Post-train Tests

Now we performed post-training tests on the saved sklearn models with model invariant tests, model directional expectation tests, and model evaluation to ensure a satisfactory level of performance. We can select a single row to run our tests. For example, let's look at the 292nd row.

```
titanic = datasets.get_titanic()
titanic.iloc[291
```

```
Out[20]: Survived    1.000000
         Pclass     -1.321707
         Sex        -1.191964
         Age         0.537055
         SibSp      -0.460589
         Parch      -0.441409
         Ticket     -0.759871
         Fare        0.689423
         Embarked   -0.604770
         Name: 61, dtype: float64
```

Figure 50: Values for row 292

(a) Invariant Tests

First, we check for invariance, by keeping everything constant and changing one irrelevant feature at a time. We should not expect the survival probability to change due to the ticket number, or port of embarkation. If it runs without error then it means all the models pass this test.

```
print("Invariant Testing:\n")
for i in os.listdir("./models/titanic/sklearn/"):
    model = pickle.load(open("./models/titanic/sklearn/"+i, 'rb'))
    print(model)
    X = datasets.get_titanic().iloc[291]
    y = X["Survived"]
    X = X[1:]
```

```
    p2_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 1.0
    X['Embarked'] = 2.47593535
    p2_embarked_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 1.0
    assert p2_prob == p2_embarked_prob
    # Change ticket number
    X['Ticket'] = 1.86005416
    p2_ticket_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 1.0
    assert p2_prob == p2_ticket_prob
print("Invariant Test Successful!")
```

(b) Directional Expectation Tests

Now, we can check for relevant feature changes for directional expectations. We should expect females to have a higher survival rate than males. Higher-class passengers have a higher survival probability. More expensive fare passengers have a higher survival probability. The following code snippet tests this functionality

```
print("Directional Expectaiton Testing:")
print("Directional Expectaiton Testing:\n")
for i in os.listdir("./models/titanic/sklearn/"):
    model = pickle.load(open("./models/titanic/sklearn/"+i, 'rb'))
    print(model)
    X = datasets.get_titanic().iloc[291]
    y = X["Survived"]
    X = X[1:]
    p2_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 1.0
    X['Sex'] = 0.83739228  #Change gender
    p2_male_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.56
    assert p2_prob > p2_male_prob,
    'Changing gender from female to male should decrease survival probability.'
    X['Pclass'] = 0.95828974 # Change class
    p2_class_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.0
    assert p2_prob > p2_class_prob,
    'Changing class from 1 to 3 should decrease survival probability.'
    X['Fare'] = -0.575978 # # Lower fare
    p2_fare_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.85
    assert p2_prob > p2_fare_prob,
    'Reducing fare should decrease survival probability.'
print("Directional expectation test successful")
```

In some cases, the directional expectation test fails for linear SVM, Gradient Boosting, and Gaussian Naive Bayes models upon a change made to relevant features. However, rarely the test runs successfully.

```
Testing model: LinearSVC(C=0.0001, max_iter=10, random_state=0, tol=0.01)

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_38917/3038047054.py in <cell line: 2>()
      9     X['Sex'] = 0.83739228  #Change gender
     10     p2_male_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.56
---> 11     assert p2_prob > p2_male_prob, 'Changing gender from female to male should decrease survival probab
ility.'
     12     X['Pclass'] = 0.95828974 # Change class
     13     p2_class_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.0

AssertionError: Changing gender from female to male should decrease survival probability.
```

```
Testing model: GradientBoostingClassifier(learning_rate=0.01, max_depth=2, max_features='auto',
                           random_state=0)

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_38917/1720687962.py in <cell line: 2>()
     12     X['Pclass'] = 0.95828974 # Change class
     13     p2_class_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.0
---> 14     assert p2_prob > p2_class_prob, 'Changing class from 1 to 3 should decrease survival probability.'
     15     X['Fare'] = -0.575978 # # Lower fare
     16     p2_fare_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.85

AssertionError: Changing class from 1 to 3 should decrease survival probability.

Testing model: GradientBoostingClassifier(learning_rate=0.01, max_depth=2, max_features='auto',
                           random_state=0)

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_38917/2639728906.py in <cell line: 2>()
     15     X['Fare'] = -0.575978 # # Lower fare
     16     p2_fare_prob = model.predict(np.array(X).reshape(1, -1))[0]  # 0.85
---> 17     assert p2_prob > p2_fare_prob, 'Reducing fare should decrease survival probability.'
     18 print("Directional expectation test succesful")

AssertionError: Reducing fare should decrease survival probability.
```

Figure 51: Directional expectation failure cases

(c) Model Evaluation to Ensure Satisfactory Performance

We evaluate our models to ensure that performance does not degrade. Here we assess model performance in terms of accuracy and it should not be less than 80% for the specific instance given.

```
print("Satisfactory performance testing")
for i in os.listdir("./models/titanic/sklearn/"):
    model = pickle.load(open("./models/titanic/sklearn/"+i,'rb'))
    print(model)
    X = datasets.get_titanic().iloc[291]
    y = X["Survived"]
    X =X[1:]
    p2_prob = model.predict(np.array(X).reshape(1,-1))[0] #1.0
```

```
    #print(p2_prob)
    acc_test =accuracy_score([y],[np.round(p2_prob)])
    #print(acc_test)
    assert acc_test > 0.8 ,"Accuracy on the test should be > 0.80"
print("Performance testing successful!")
```

6.6 Limitations

(a) Limitations of Novel Testing Approaches

Metamorphic testing is the foundation of many testing approaches for non-testable programs such as machine learning. However, it has certain limitations. First, for complex problems, it is not always easy to formulate a metamorphic relation between the input and output components. Second, all non-testable programs cannot be tested with metamorphic testing because not all necessary properties are metamorphic relations. Third, an exhaustive list of all possible metamorphic relations for even medium-scale ML applications is less feasible and costly in terms of computational resources. In a fourth step, systematic metamorphic relation identification is still in its infancy. Finally, comprehensive empirical studies for a unified understanding of MT and a thorough evaluation of MT's overall effectiveness in various application domains are still not yet matured.

Behavioral testing is a recent approach and has gained popularity among ML enthusiasts. However, identifying model capabilities is challenging. A simple sentiment analysis model, for instance, would involve identifying all possible language capabilities. This requires a thorough understanding of human language, and matching capabilities to a sentiment model is challenging. Using a checklist for testing models is against the inherent foundation of machine learning. The detailed requirement gathering and the design architecture of such systems are not feasible due to data entanglement and data cascade issues.

The application of smoke testing to machine learning appears easy. However, the design and generation of test cases using equivalence classes and boundary value analysis even for simple classification problems are computationally expensive and less practical. In addition, smoke testing does not provide detailed characteristics of model behavior. This is because smoke tests check the core functionality of the system: prediction for classification.

(b) Prototype Limitation and Future Work

In the future, we want to focus on adding robustness testing. The problem can be handled by identifying model capabilities and data features and performing full post-train tests with minimum functionality tests, invariant tests, and directional expectation tests. Also, we plan on considering more tests on the pre-train tests for the model, in order to include a variety of data features in order to improve the quality of the data as well as the model prediction. In addition, we also plan to orchestrate the machine-learning workflow with MLflow to properly track the appropriate hyperparameters, cleaned data, visualizations, and other artifacts to make the application robust and reproducible.

7 Conclusion

In this thesis, we performed an analysis of machine learning bugs. We performed this using AI and ML practitioner posts from Stack Overflow and GitHub and using two static analysis tools i.e SonarcDeepsource. This helped us to better equip ourselves with frequent bugs in open-source ML systems. We proved that bugs in open-source machine learning systems can exist in the data, the learning program, or the ML framework. The decision logic of ML systems is dependent on data, and the non-deterministic nature of the algorithms makes testing such systems with traditional software testing approaches challenging. Due to this, we explored and examined novel testing techniques: smoke testing, metamorphic testing, and model behavioral testing. By using these techniques, we are able to replicate already existing bugs.

In an effort to discover previously undisclosed bugs, minimize possible bugs at an early stage, and find discrepancies, we implemented a machine-learning prototype to examine and evaluate seventeen classification algorithms from Scikit-learn, PySpark, and Keras. From the empirical experiments, no observable differences were found to provide evidence of a previously undiscovered bug, and also the variation in performance of the models from varying frameworks is negligible.

In particular, we contributed the following core point to our fellow machine learning enthusiasts, users and researchers.

(a) We systematically analyzed representative examples of frequently occurring bugs in open-source machine-learning systems. This was done by systematic reviews of developer posts, performing static data analysis, and replication of representative work on novel testing techniques.

(b) In an effort to discover previously undisclosed bugs and possible discrepancies, we designed and developed a proof of concept to evaluate seventeen classification algorithms from three open-source libraries

(c) We performed model and data validation using Deepcheck to check for possible model and data drifts between the training and test sets.

(d) We implemented model pre-training tests and data validation to minimize bugs. In addition, We applied a systematic and similar data preprocessing method

with some slight differences for the pyspark and scikit-learn libraries in order to improve data quality and model robustness.

(e) We provide a uniform way of data preparation, model training and evaluation, hyperparameter tuning, and result visualization using representative performance metrics. We have used Keras in both libraries to improve average accuracy, otherwise, accuracy was varying a lot. As a result of the data inherent preprocessing, the same hyperparameters, and other model settings, the values of the various performance metrics are consistent.

(f) We implemented model post-train tests to ensure learned behavior works as expected. We demonstrated this using invariant testing and directional expectation tests using the scikit-learn saved models for a representative train instance. In addition, we tested for performance degradation of the saved models to ensure satisfactory performance.

Bibliography

[1] Artificial intelligence incident database, https://incidentdatabase.ai/

[2] Building machine learning pipelines, https://www.oreilly.com/library/view/building-machine-learning/9781492053187/

[3] Choosing the right estimator, https://scikit-learn/stable/tutorial/machine_learning_map/index.html

[4] Codescene: Software engineering intelligence, https://codescene.com

[5] Common vulnerabilities and exposures, https://www.cve.org/

[6] International software testing qualifications board, https://www.istqb.org/

[7] K means clustering. https://sakshi-tajane.medium.com/k-means-clustering-clustering-is-the-classification-of-objects-into-different- accessed: 2022-07-27

[8] Kernel functions. https://towardsdatascience.com/kernel-function-6f1d2be6091, accessed: 2022-07-25

[9] Quality assurance and testing the ml model, https://dzone.com/articles/quality-assurancetesting-the-machine-learning-mode

[10] sklearn.linear_model.logisticregression-scikit-learn 1.1.1 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html, accessed: 2022-07-25

[11] Types of machine learning, https://www.coursera.org/articles/types-of-machine-learning

[12] Ieee standard glossary of software engineering terminology. IEEE Std 610.12-1990 pp. 1–84 (1990)

[13] Metamorphic testing for machine learning models (August 2018), https://vitalflux.com/qa-metamorphic-testing-machine-learning/

[14] Auc-roc curve in machine learning clearly explained (June 2020), https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/

[15] Evaluation metrics for machine learning for data scientists (October 2020), https://www.analyticsvidhya.com/blog/2020/10/quick-guide-to-evaluation-metrics-for-supervised-and-unsupervised-machine-lear

[16] Acharya, A.: In-depth guide to ml model debugging and tools you need to know (July 2021), https://neptune.ai/blog/ml-model-debugging-and-tools

[17] Al Alamin, M.A., Uddin, G.: Quality assurance challenges for machine learning software applications during software development life cycle phases. In: 2021 IEEE International Conference on Autonomous Systems (ICAS). pp. 1–5. IEEE (2021)

[18] Al-Rubaie, M., Chang, J.M.: Privacy-preserving machine learning: Threats and solutions. IEEE Security & Privacy 17(2), 49–58 (2019)

[19] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., Zimmermann, T.: Software engineering for machine learning: A case study. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 291–300. IEEE (2019)

[20] Arnold, M., Boston, J., Desmond, M., Duesterwald, E., Elder, B., Murthi, A., Navratil, J., Reimer, D.: Towards automating the ai operations lifecycle. arXiv preprint arXiv:2003.12808 (2020)

[21] Bandy, J., Vincent, N.: Addressing "documentation debt" in machine learning research: A retrospective datasheet for bookcorpus. arXiv preprint arXiv:2105.05241 (2021), https://arxiv.org/abs/2105.05241

[22] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE transactions on software engineering 41(5), 507–525 (2014)

[23] Bhardwaj, A.: Silhouette coefficient : Validating clustering techniques (May 2020), https://towardsdatascience.com/silhouette-coefficient-validating-clustering-techniques-e976bb81d10c

[24] Bond, A.M., Zhang, J., Gundry, L., Kennedy, G.F.: Opportunities and challenges in applying machine learning to voltammetric mechanistic studies. Current Opinion in Electrochemistry p. 101009 (2022)

[25] Breck, E., Cai, S., Nielsen, E., Salib, M., Sculley, D.: The ml test score: A rubric for ml production readiness and technical debt reduction. In: 2017 IEEE International Conference on Big Data (Big Data). pp. 1123–1132. IEEE (2017)

[26] Brownlee, J.: Master Machine Learning AlgorithmsDiscover How They Work and Implement Them From Scratch (2016)

[27] CHAKRABORTY, K.: A complete guide to support vector machines (June 2019), https://medium.com/@kushaldps1996/a-complete-guide-to-support-vector-machines-svms-501e71aec19e

[28] Chaudhary, A.: An overview of the "checklist" framework for fine-grained evaluation of nlp models (July 2020), https://amitness.com/2020/07/checklist/

[29] Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. arXiv preprint arXiv:2002.12543 (2020)

[30] Chen, T.Y., Kuo, F.C., Liu, H., Poon, P.L., Towey, D., Tse, T., Zhou, Z.Q.: Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys (CSUR) 51(1), 1–27 (2018)

[31] Chiye, E.H.: Conventional software vs. machin learning application from a tester's perspective (July 2020), https://medium.com/@hockchiye.er/conventional-software-vs-machine-learning-application-a-testers-perspective-7b

[32] Deepchecks: Testing machine learning models. https://deepchecks.com/ (2022)

[33] DeepSource: Deepsource: Fast and reliable static analysis platform. https://deepsource.io/ (2022)

[34] Ding, F., Hardt, M., Miller, J., Schmidt, L.: Retiring adult: New datasets for fair machine learning. Advances in Neural Information Processing Systems 34, 6478–6490 (2021)

[35] Ding, J., Hu, X.H., Gudivada, V.: A machine learning based framework for verification and validation of massive scale image data. IEEE Transactions on Big Data 7(2), 451–467 (2017)

[36] Dwarakanath, A., Ahuja, M., Sikand, S., Rao, R.M., Bose, R.J.C., Dubash, N., Podder, S.: Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In: Proceedings of the 27th

ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 118–128 (2018)

[37] Felderer, M., Ramler, R.: Quality assurance for ai-based systems: Overview and challenges (introduction to interactive session). In: International Conference on Software Quality. pp. 33–42. Springer (2021)

[38] Gebru, T., Morgenstern, J., Vecchione, B., Vaughan, J.W., Wallach, H., Iii, H.D., Crawford, K.: Datasheets for datasets. Communications of the ACM 64(12), 86–92 (2021)

[39] Grigorev, A.: Machine learning bookcamp: build a portfolio of real-life projects. Manning, Shelter Island (2021)

[40] Habib, M.T., Majumder, A., Nandi, R.N., Ahmed, F., Uddin, M.S.: A comparative study of classifiers in the context of papaya disease recognition. In: Uddin, M.S., Bansal, J.C. (eds.) Proceedings of International Joint Conference on Computational Intelligence. pp. 417–429. Springer Singapore, Singapore (2020)

[41] Herbold, S., Haar, T.: Smoke testing for machine learning: Simple tests to discover severe defects. arXiv preprint arXiv:2009.01521 (2020)

[42] Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P.: Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1110–1121 (2020)

[43] Hynes, N., Sculley, D., Terry, M.: The data linter: Lightweight, automated sanity checking for ml data sets. In: NIPS MLSys Workshop. vol. 1 (2017)

[44] Ishikawa, F., Yoshioka, N.: How do engineers perceive difficulties in engineering of machine-learning systems?-questionnaire survey. In: 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP). pp. 2–9. IEEE (2019)

[45] Islam, M.J., Nguyen, G., Pan, R., Rajan, H.: A comprehensive study on deep learning bug characteristics. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 510–520 (2019)

[46] Islam, M.J., Nguyen, H.A., Pan, R., Rajan, H.: What do developers ask about ml libraries? a large-scale study using stack overflow. arXiv preprint arXiv:1906.11940 (2019)

[47] Jordan, J.: Effective testing for machine learning systems. (2020), https://www.jeremyjordan.me/testing-ml/

[48] Kanewala, U., Bieman, J.M.: Using machine learning techniques to detect metamorphic relations for programs without test oracles. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 1–10. IEEE (2013)

[49] Kanstrén, T.: Metamorphic testing of machine learning based systems (November 2020), https://towardsdatascience.com/metamorphic-testing-of-machine-learning-based-systems-e1fe13baf048

[50] Khan, K., Rehman, S.U., Aziz, K., Fong, S., Sarasvady, S.: Dbscan: Past, present and future. In: The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014). pp. 232–238. IEEE (2014)

[51] Kjetså, T.I.S.: MLOps-challenges with Operationalizing Machine Learning Systems. Master's thesis, Norwegian University of Science and Technology (2021)

[52] Kumar, M., Rath, S.: Chapter 15 - feature selection and classification of microarray data using machine learning techniques. In: Tran, Q.N., Arabnia, H.R. (eds.) Emerging Trends in Applications and Infrastructures for Computational Biology, Bioinformatics, and Systems Biology, pp. 213–242. Emerging Trends in Computer Science and Applied Computing, Morgan Kaufmann, Boston (2016), https://www.sciencedirect.com/science/article/pii/B9780128042038000158

[53] Ma, L., Juefei-Xu, F., Xue, M., Hu, Q., Chen, S., Li, B., Liu, Y., Zhao, J., Yin, J., See, S.: Secure deep learning engineering: A software quality assurance perspective. arXiv preprint arXiv:1810.04538 (2018)

[54] Marijan, D., Gotlieb, A.: Software testing for machine learning. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 13576–13582 (2020)

[55] Marijan, D., Gotlieb, A., Ahuja, M.K.: Challenges of testing machine learning based systems. In: 2019 IEEE international conference on artificial intelligence testing (AITest). pp. 101–102. IEEE (2019)

[56] Masuda, S., Ono, K., Yasue, T., Hosokawa, N.: A survey of software quality for machine learning applications. In: 2018 IEEE International conference on software testing, verification and validation workshops (ICSTW). pp. 279–284. IEEE (2018)

[57] McGregor, S.: Preventing repeated real world ai failures by cataloging incidents: The ai incident database. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 15458–15463 (2021)

[58] Michael Steinbach, P.N.T., Kumar, V.: Introduction to Data Mining. Pearson Education (2014)

[59] Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I.D., Gebru, T.: Model cards for model reporting. In: Proceedings of the conference on fairness, accountability, and transparency. pp. 220–229 (2019)

[60] Mitchell, T.M., Mitchell, T.M.: Machine learning, vol. 1. McGraw-hill New York (1997)

[61] Mohandas, G.: Mlops course - made with ml, https://madewithml.com/courses/mlops/

[62] Murphy, C., Kaiser, G.E., Hu, L.: Properties of machine learning applications for use in metamorphic testing (2008)

[63] Murphy, K.P.: Machine Learning: A Probabilistic Perspective. Adaptive Computation and Machine Learning series, MIT Press, Cambridge, MA, USA (August 2012)

[64] Navlani, A.: Naive Bayes Classification using Scikit-learn, https://www.datacamp.com/tutorial/naive-bayes-scikit-learn

[65] Neumann, P.G.: Forum on risks to the public in computers and related systems. The RISKS Digest http://catless.ncl.ac.uk/Risks/

[66] Nighania, K.: Various ways to evaluate a machine learning models performance (January 2019), https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-modelsperformance-230449055f15

[67] Oladele, S.: Model deployment challenges: 6 lessons from 6 ml engineers (December 2021), https://neptune.ai/blog/model-deployment-challenges-lessons-from-ml-engineers

[68] Papernot, N., McDaniel, P., Sinha, A., Wellman, M.: Towards the science of security and privacy in machine learning. arXiv preprint arXiv:1611.03814 (2016)

[69] Patel, H.: Model debugging strategies: Machine learning guide (June 2021), https://neptune.ai/blog/model-debugging-strategies-machine-learning

[70] R, V.: Beyondaccuracy: Behavioral testing of nlp models with checklist (April 2022), https://medium.com/read-a-paper/beyond-accuracy-behavioral-testing-of-nlp-models-with-checklist-read-a-paper-2

[71] Raschka, S.: Model evaluation, model selection, and algorithm selection in machine learning. arXiv preprint arXiv:1811.12808 (2018)

[72] Ribeiro, M.T., Wu, T., Guestrin, C., Singh, S.: Beyond accuracy: Behavioral testing of nlp models with checklist. In: Association for Computational Linguistics (ACL) (2020)

[73] Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., Tonella, P.: Testing machine learning based systems: a systematic mapping. Empirical Software Engineering 25(6), 5193–5254 (2020)

[74] Sambasivan, N., Kapania, S., Highfill, H., Akrong, D., Paritosh, P., Aroyo, L.M.: "everyone wants to do the model work, not the data work": Data cascades in high-stakes ai. In: proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. pp. 1–15 (2021)

[75] Schott, L., Rauber, J., Bethge, M., Brendel, W.: Towards the first adversarially robust neural network model on mnist. arXiv preprint arXiv:1805.09190 (2018)

[76] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M.: Machine learning: The high interest credit card of technical debt (2014)

[77] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. Advances in neural information processing systems 28 (2015)

[78] Seca, D.: A review on oracle issues in machine learning. Tech. Rep. arXiv:2105.01407, arXiv (May 2021), http://arxiv.org/abs/2105.01407

[79] Serban, A., van der Blom, K., Hoos, H., Visser, J.: Adoption and effects of software engineering best practices in machine learning. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 1–12 (2020)

[80] Shai Shalev-Shwartz, S.B.D.: Understanding Machine Learning From Theory to Algorithms. Cambridge University Press (2014)

[81] Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Sarro, F.: A survey on machine learning techniques for source code analysis. arXiv preprint arXiv:2110.09610 (2021)

[82] Siebert, J., Joeckel, L., Heidrich, J., Trendowicz, A., Nakamichi, K., Ohashi, K., Namba, I., Yamamoto, R., Aoyama, M.: Construction of a quality model for machine learning systems. Software Quality Journal 30(2), 307–335 (2022)

[83] Sonarqube.org: Code quality and code security. `https://www.sonarqube.org/` (2022)

[84] Sugali, K.: Software testing: Issues and challenges of artificial intelligence machine learning (2021)

[85] Sultanow, E., Ullrich, A., Konopik, S., Vladova, G.: Machine learning based static code analysis for software quality assurance. In: 2018 Thirteenth International Conference on Digital Information Management (ICDIM). pp. 156–161. IEEE (2018)

[86] Team, G.L.: Regression testing in software testing (June 2021), `https://www.mygreatlearning.com/blog/regression-testing/`

[87] Wan, Z., Xia, X., Lo, D., Murphy, G.C.: How does machine learning change software development practices? IEEE Transactions on Software Engineering 47(9), 1857–1871 (2019)

[88] Wirth, R., Hipp, J.: Crisp-dm: Towards a standard process model for data mining. In: Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining. vol. 1, pp. 29–39. Manchester (2000)

[89] Xie, X., Ho, J.W., Murphy, C., Kaiser, G., Xu, B., Chen, T.Y.: Testing and validating machine learning classifiers by metamorphic testing. Journal of Systems and Software 84(4), 544–558 (2011)

[90] Yan, E.: How to test machine learning code and systems (September 2020), https://eugeneyan.com/writing/testing-ml/

[91] Yan, E.: How to test machine learning code and systems (September 2020), https://eugeneyan.com/writing/testing-ml/

[92] Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering (2020)

[93] Zhou, Z.Q., Huang, D.H., Tse, T.H., Yang, Z., Huang, H., Chen, T.Y.: Metamorphic testing and its applications. In: PROCEEDINGS OF THE 8TH INTERNATIONAL SYMPOSIUM ON FUTURE SOFTWARE TECHNOLOGY (ISFST 2004). Software Engineers Association (2004)

Appendix A

7.1 Dataset Dictionary

(a) Titanic Disaster Dataset

The Titanic's sinking has become one of history's most infamous shipwrecks. On April 15, 1912, during her maiden voyage, the widely considered **"unsinkable"** Titanic sank after colliding with an iceberg. While there was some element of luck involved in surviving, it seems some groups of people were more likely to survive than others. The data describes the survival status of individual passengers and contain 891 unique training and 418 test sets.

- **Survived (class variable)**: the survival status of passengers, 0 = No, 1 = Yes
- **Pclass**: refers to passenger class (1st, 2nd, 3rd), and is a proxy for socio-economic class.
- **SibSp**: The number of siblings i.e brother, sister, stepbrother, stepsister
- **Parch**: The number of parents aboard the Titanic
- **Ticket**: Ticket number
- **Fare**: passenger fare
- **Cabin**: cabin number
- **Embarked**: port of embarkation, C = Cherbourg, Q = Queenstown, S = Southampton
- **Name**, **PassengerId**, **Sex**, and **Age** are self-explanatory

The dataset contains categorical features such as Survived, Sex, Pclass, and embarked. Age, Fare. Discrete, SibSp and Parch are numerical features.

(b) Fashion MNIST Dataset

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Each row is a separate image, column 1 is the class label, and the remaining columns are pixel

numbers (784 total). The dataset is a novel image dataset for benchmarking machine learning algorithms

(c) Pima Indian Diabetes Dataset

The dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the data. These instances were selected from a larger database under several constraints. All of the patients here are females who are at least 21 years old and of Pima Indian descent.

The dataset contains 768 instances with all numeric values

- **Pregnancies**: Number of times pregnant
- **Glucose**: Plasma glucose concentration a 2 hours in an oral glucose tolerance test
- **BloodPressure**: Diastolic blood pressure (mm Hg)
- **SkinThickness**: Triceps skin fold thickness (mm)
- **Insulin**: 2-Hour serum insulin (mu U/ml)
- BMI: Body mass index
- **DiabetesPedigreeFunction**: Diabetes pedigree function
- **Age**: Age (years)
- **Outcome**: Class variable (0 or 1)**268** of **768** are 1, the others are 0

7.2 Further Details on the Solution Approach

(a) Data Processing Logic for Titanic Dataset

```
def get_titanic(source="train.csv"):
    titanic = pd.read_csv(source)
    titanic.drop(columns="Cabin", inplace=True)
    titanic["Age"][titanic["Name"].str.contains("Mr.")
    & ~titanic["Sex"].str.contains("female") & titanic["Age"].isna()] = \
        round(titanic["Age"][titanic["Name"].str.contains("Mr.")
     & ~titanic["Sex"].str.contains("female")].mean())
```

```
    titanic["Age"][titanic["Name"].str.contains("Master.")
      & ~titanic["Sex"].str.contains("female") & titanic["Age"].isna()] = \
        round(titanic["Age"][titanic["Name"].str.contains(
            "Master.") & ~titanic["Sex"].str.contains("female")].mean())
    titanic["Age"][titanic["Name"].str.contains("Mrs.")
     & titanic["Sex"].str.contains("female") & titanic["Age"].isna()] = \
        round(titanic["Age"][titanic["Name"].str.contains("Mrs.")
              & titanic["Sex"].str.contains("female")].mean())
    titanic["Age"][titanic["Name"].str.contains("Miss.")
    & titanic["Sex"].str.contains("female") & titanic["Age"].isna()] = \
        round(titanic["Age"][titanic["Name"].str.contains("Miss.")
              & titanic["Sex"].str.contains("female")].mean())
    titanic["Age"][~titanic["Name"].str.contains("Miss.")
     & ~titanic["Name"].str.contains("Mr.") & ~titanic["Name"]
        .str.contains("Master.") & titanic["Age"].isna()] = \
        round(titanic["Age"][~titanic["Name"].str.contains("Miss.")
            & ~titanic["Name"].str.contains(
            "Mr.") & ~titanic["Name"].str.contains("Master.")].mean())

    titanic["Embarked"][titanic["Embarked"].isna()] = "S"
    titanic.drop(columns=["PassengerId", "Name"], inplace=True)
    titanic.drop(
        index=titanic[titanic.drop(columns="Survived").duplicated()].index,
        inplace=True)
    titanic["Sex"] = np.where(titanic["Sex"] == "male", 1, 0)
    titanic["Ticket"].replace(list(titanic["Ticket"].value_counts().index),
                              range(len(titanic["Ticket"].value_counts())),
                              inplace=True)
    titanic["Embarked"].replace(list(titanic["Embarked"].value_counts().index),
                                range(len(titanic["Embarked"].value_counts())),
                                inplace=True)
    add = titanic.drop(columns="Survived").sum(axis=1)
    titanic = titanic[(add < add.quantile(0.9)) & (add > add.quantile(0.1))]
    titanic = pd.concat([titanic[titanic['Survived'] == 0]
     .sample(len(titanic[titanic['Survived'] == 1])),
                         titanic[titanic['Survived'] == 1]], axis=0)
    col = list(titanic.columns)
    col.remove("Survived")
    for i in col:
        titanic[i] = (titanic[i] - titanic[i].mean()) / titanic[i].std()
    return titanic
```

```
49
50
51  def get_sklearn_titanic(source="train.csv", test_size=0.2, seed=0):
52          titanic = get_titanic(source)
53          titanic_label = titanic["Survived"]
54          titanic.drop(columns="Survived", inplace=True)
55          return train_test_split(titanic, titanic_label, test_size=test_size,
56          random_state=seed)
57  def get_pyspark_titanic(sc, source="train.csv", test_size=0.2, seed=0):
58          titanic = get_titanic(source)
59          titanic = titanic.rename(columns={'Survived': 'label'})
60          sqlContext = SQLContext(sc)
61          data = sqlContext.createDataFrame(titanic)
62          col = list(titanic.columns)
63          col.remove("label")
64          va = VectorAssembler(inputCols = col, outputCol='features')
65          va_df = va.transform(data)
66          va_df = va_df.select(['features', 'label'])
67          return va_df.randomSplit([1-test_size, test_size], seed=seed)
68
69
```

(b) Data Processing logic for Diabetes Dataset

```
1   def get_diabetes(source="diabetes.csv", seed=0):
2           diabetes = pd.read_csv(source)
3
4        assert set(diabetes.columns) == set(("Pregnancies", "Glucose",
5       "BloodPressure", "SkinThickness", "Insulin", "BMI",
6       "DiabetesPedigreeFunction", "Age", "Outcome")), "Unexpected Column(s)"
7           assert diabetes[diabetes.columns].isna().sum().sum() == 0,
8           "Missing values found"
9           assert set(diabetes["Outcome"].unique()) == set((0, 1)),
10          "Unknown outcome value recieved"
11          assert (diabetes[diabetes.columns].min() >= 0).all(),
12          "All values should be positive"
13
14          col = list(diabetes.columns)
15          col.remove("Outcome")
16          for i in col:
17                  diabetes[i] = (diabetes[i] - diabetes[i].mean())
```

```
18            / diabetes[i].std()
19        add = diabetes.drop(columns="Outcome").sum(axis=1)
20        diabetes = diabetes[(add < add.quantile(0.95))
21        & (add > add.quantile(0.05))]
22        diabetes = pd.concat([diabetes[diabetes['Outcome'] == 0]
23    .sample(len(diabetes[diabetes['Outcome'] == 1]), \
24    random_state=seed), diabetes[diabetes['Outcome'] == 1]], axis=0)
25        return diabetes
26
27
28 def get_sklearn_diabetes(source="diabetes.csv", test_size=0.2, seed=0):
29        diabetes = get_diabetes(source)
30        diabetes_label = diabetes["Outcome"]
31        diabetes.drop(columns="Outcome", inplace=True)
32        return train_test_split(diabetes, diabetes_label,
33    test_size=test_size, random_state=seed)
34
35 def get_pyspark_diabetes(sc, source="diabetes.csv",test_size=0.2, seed=0):
36        diabetes = get_diabetes(source)
37        diabetes = diabetes.rename(columns={'Outcome': 'label'})
38        sqlContext = SQLContext(sc)
39        data = sqlContext.createDataFrame(diabetes)
40        col = list(diabetes.columns)
41        col.remove("label")
42        va = VectorAssembler(inputCols = col, outputCol='features')
43        va_df = va.transform(data)
44        va_df = va_df.select(['features', 'label'])
45        return va_df.randomSplit([1-test_size, test_size], seed=seed)
```

(c) Data Processing logic for Fashion MNIST dataset

```
1 def get_fashion(source="fashion-mnist_train.csv"):
2        fashion = pd.read_csv(source)
3        #fashion = fashion[:500]
4
5        assert fashion[fashion.columns].isna().sum().sum() == 0,
6    "Missing values found"
7        assert set(fashion["label"].unique()) ==
8    set((0, 1, 2, 3, 4, 5, 6, 7, 8, 9)),
9    "Unknown label recieved"
```

```
10          assert (fashion[fashion.columns].min() >= 0).all(),
11      "All values should be positive"
12          assert (fashion[fashion.columns].max() <= 255).all(),
13      "All values should be less than 256"
14
15          col = list(fashion.columns)
16          col.remove("label")
17          for i in col:
18                  if ((fashion[i]>0).sum())<25000:
19                      fashion.drop(columns=i, inplace=True)
20          fashion.drop(index=fashion[fashion.drop(columns="label").
21      duplicated()].index, inplace=True)
22          col = list(fashion.columns)
23          col.remove("label")
24          for i in col:
25                  fashion[i] = fashion[i] / 255
26          new = fashion.iloc[0:2]
27          new.drop(index=[0,1], inplace=True)
28          for i in range(10):
29                  add = fashion[fashion["label"]==i].drop(columns="label")
30             .sum(axis=1)
31                  new = pd.concat([new, fashion[fashion["label"]==i]
32      [(add < add.quantile(0.99)) & (add > add.quantile(0.01))]], axis=0)
33          fashion = new.sort_index()
34          pca = PCA(n_components=10)
35          new = pca.fit_transform(fashion.drop(columns="label"))
36          new = pd.DataFrame(data = new,
37      columns = ['C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'C10'],
38          index=fashion.index)
39          for i in new.columns:
40                  new[i] = (new[i] - new[i].mean()) / new[i].std()
41          fashion = pd.concat([fashion["label"], new], axis=1)
42          return fashion
43
44
45  def get_sklearn_fashion(source="fashion-mnist_train.csv",
46      test_size=0.2, seed=0):
47          fashion = get_fashion(source)
48          fashion_label = fashion["label"]
49          fashion.drop(columns="label", inplace=True)
50          return train_test_split(fashion, fashion_label,
```

```
    test_size=test_size,random_state=seed)

def get_pyspark_fashion(sc, source="fashion-mnist_train.csv",
    test_size=0.2, seed=0):
        fashion = get_fashion(source)
        sqlContext = SQLContext(sc)
        data = sqlContext.createDataFrame(fashion)
        col = list(fashion.columns)
        col.remove("label")
        va = VectorAssembler(inputCols = col, outputCol='features')
        va_df = va.transform(data)
        va_df = va_df.select(['features', 'label'])
        return va_df.randomSplit([1-test_size, test_size], seed=seed)
```

(d) Model Evaluation and Visualization Method

We implemented a method to evaluate and visualize the accuracy, precision, recall, roc, and various associated metrics.

```
def eval(clf, real, pred, ax, bx):
        accuracy = (pred == real).mean()
        confusion = confusion_matrix(pred, real)
        try:
                roc = roc_auc_score(real, pred)
        except:
                roc = None
        precision = precision_score(real, pred, average="macro")
        recall = recall_score(real, pred, average="macro")
        if len(unique(real))==2:
                RocCurveDisplay.from_predictions(real, pred)
            .plot(ax=ax, name=clf)
                PrecisionRecallDisplay.from_predictions(real, pred)
              .plot(ax=bx, name=clf)
        else:
                pass
        return accuracy, confusion, roc, precision, recall

def plot(sklearn, pyspark, metric):
        models = ["LinearSVC", "Logistic Regression", "Decision Tree",
        "Random Forest", "Naive Bayes", "Gradient Boosting",
```

```
        "MLP Classifier", "One vs Rest", "Keras Network"]
        plt.figure(figsize=(20,8))
        plt.bar(models, sklearn, align="edge", width=-0.3)
        plt.bar(models, pyspark, align="edge", width=0.3)
        plt.legend(["Sklearn "+metric, "Pyspark "+metric], loc="lower right")
        plt.title(metric, fontsize=16)
        plt.grid()

```

(e) Define, Train, Evaluate, and Visualize Baseline Models

```

# Define baseline models
sklearn_classifiers = [
    sklearn_models.LinearSVC(random_state=seed),
    sklearn_models.LogisticRegression(random_state=seed),
    sklearn_models.DecisionTreeClassifier(random_state=seed),
    sklearn_models.RandomForestClassifier(random_state=seed),
    sklearn_models.GaussianNB(),
    sklearn_models.GradientBoostingClassifier(random_state=seed),
    sklearn_models.MLPClassifier(random_state=seed),
    sklearn_models.OneVsRestClassifier(sklearn_models.LinearSVC())]
pyspark_classifiers = [
    pyspark_models.LinearSVC(labelCol="label"),
    pyspark_models.LogisticRegression(labelCol="label"),
    pyspark_models.DecisionTreeClassifier(labelCol="label", seed=seed),
    pyspark_models.RandomForestClassifier(labelCol='label', seed=seed),
    pyspark_models.NaiveBayes(labelCol='label', modelType="gaussian"),
    pyspark_models.GBTClassifier(labelCol='label', seed=seed),
    pyspark_models.MultilayerPerceptronClassifier(
        labelCol='label',seed=seed,
        layers=[
            len(test.toPandas()["features"][0]),
            100,]),
    pyspark_models.OneVsRest(labelCol='label',
                             classifier=pyspark_models.LinearSVC())]

# train, evaluate and visualise baseline models
sklearn_accuracy = []
```

```
sklearn_confusion = []
sklearn_roc = []
sklearn_precision = []
sklearn_recall = []
_, ax = plt.subplots(figsize=(20, 10))
_, bx = plt.subplots(figsize=(20, 10))
for clf in sklearn_classifiers:
    clf.fit(x_train, y_train)
    accuracy, confusion, roc, precision, recall = eval_methods.eval(
        clf, y_test, clf.predict(x_test), ax, bx)
    sklearn_accuracy.append(accuracy)
    sklearn_confusion.append(confusion)
    sklearn_roc.append(roc)
    sklearn_precision.append(precision)
    sklearn_recall.append(recall)

clf = keras.Sequential()
clf.add(keras.layers.Dense(1024, activation='relu'))
clf.add(keras.layers.Dense(512, activation='relu'))
clf.add(keras.layers.Dense(128, activation='relu'))
clf.add(keras.layers.Dense(32, activation='relu'))
clf.add(keras.layers.Dense(1, activation='sigmoid'))

clf.compile(optimizer=keras.optimizers.Adam(),
            loss=keras.losses.BinaryCrossentropy(),
            metrics=keras.metrics.BinaryAccuracy())

clf.fit(x_train, y_train, verbose=0)
accuracy, confusion, roc, precision, recall = eval_methods.eval(
    clf, y_test,
    np.round(clf.predict(x_test, verbose=0))[:, 0], ax, bx)
sklearn_accuracy.append(accuracy)
sklearn_confusion.append(confusion)
sklearn_roc.append(roc)
sklearn_precision.append(precision)
sklearn_recall.append(recall)

```

(f) Train, Evaluate, and Save Optimized PySpark Models

```
pyspark_accuracy = []
pyspark_confusion = []
pyspark_roc = []
pyspark_precision = []
pyspark_recall = []
_, ax = plt.subplots(figsize=(20, 10))
_, bx = plt.subplots(figsize=(20, 10))
path = "./models/titanic/pyspark/"
if os.path.exists(path):
    shutil.rmtree(path, ignore_errors=True)
for clf in pyspark_classifiers:
    clf = clf.setParams(**grid[str(clf)[:str(clf).find("_")]])
    clf = clf.fit(train)
    if not os.path.exists(path):
        os.makedirs(path)
    clf.save(path"+str(clf)[:str(clf).find("_")])
    real = np.array([1 if "1" in str(x) else 0 for x in
    clf.transform(test).select("label").collect()])
    pred = np.array([1 if "1" in str(x) else 0 for x in
    clf.transform(test).select("prediction").collect()])
    accuracy, confusion, roc, precision, recall =
    eval_methods.eval(clf, real, pred, ax, bx)
    pyspark_accuracy.append(accuracy)
    pyspark_confusion.append(confusion)
    pyspark_roc.append(roc)
    pyspark_precision.append(precision)
    pyspark_recall.append(recall)

clf = keras.Sequential()
clf.add(keras.layers.Dense(1024, activation='relu'))
clf.add(keras.layers.Dense(512, activation='relu'))
clf.add(keras.layers.Dense(128, activation='relu'))
clf.add(keras.layers.Dense(32, activation='relu'))
clf.add(keras.layers.Dense(1, activation='sigmoid'))

clf.compile(optimizer=keras.optimizers.Adam(),
            loss=keras.losses.BinaryCrossentropy(),
            metrics=keras.metrics.BinaryAccuracy())

clf.fit(x_train, y_train, verbose=0, epochs=5)
```

```
clf.save("./models/titanic/keras.h5")
accuracy, confusion, roc, precision, recall =
eval_methods.eval(clf, y_test, np.round(clf.predict(x_test,
verbose=0))[:,0], ax, bx)
pyspark_accuracy.append(accuracy)
pyspark_confusion.append(confusion)
pyspark_roc.append(roc)
pyspark_precision.append(precision)
pyspark_recall.append(recall)
```

Appendix B

7.1 Extended Version of Experimental Results

(a) The Deepcheck Test Results Organization

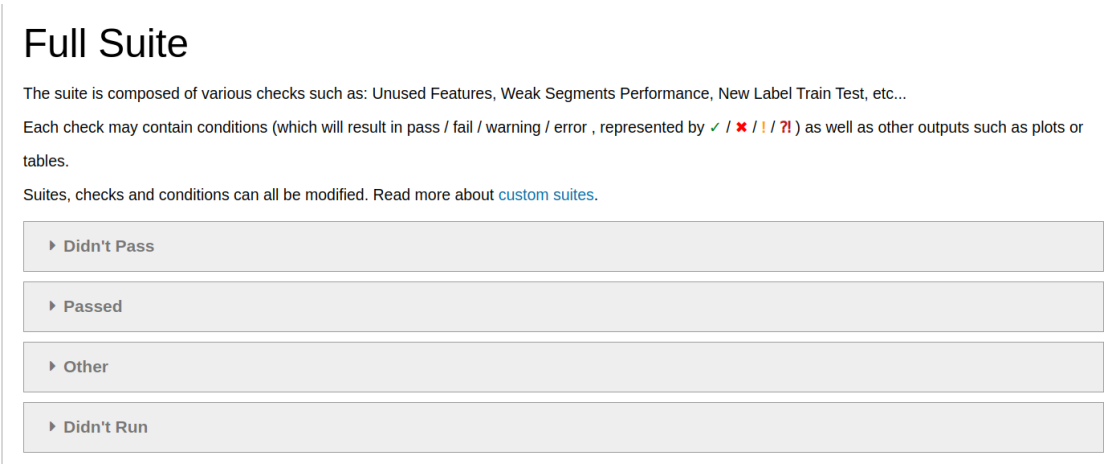

Figure 52: Deepchecks tests results organization

(b) Invariant, Directional Expectation, and Performance test results

```
Invariant Testing:

GradientBoostingClassifier(learning_rate=0.01, max_depth=2, max_features='auto',
                           random_state=0)
LogisticRegression(C=0.01, max_iter=10, random_state=0, tol=0.01)
MLPClassifier(batch_size=128, max_iter=500, random_state=0)
OneVsRestClassifier(estimator=LinearSVC(), n_jobs=1)
GaussianNB(var_smoothing=1e-07)
LinearSVC(C=0.0001, max_iter=10, random_state=0, tol=0.01)
DecisionTreeClassifier(max_depth=2, random_state=0)
RandomForestClassifier(max_depth=10, random_state=0)
Invariant testing successful
```

```
Directional Expectaiton Testing:

GradientBoostingClassifier(learning_rate=0.01, max_depth=2, max_features='auto',
                           random_state=0)
LogisticRegression(C=0.01, max_iter=10, random_state=0, tol=0.01)
MLPClassifier(batch_size=128, max_iter=500, random_state=0)
OneVsRestClassifier(estimator=LinearSVC(), n_jobs=1)
GaussianNB(var_smoothing=1e-07)
LinearSVC(C=0.0001, max_iter=10, random_state=0, tol=0.01)
DecisionTreeClassifier(max_depth=2, random_state=0)
RandomForestClassifier(max_depth=10, random_state=0)
Directional Exepction test successful
```

```
Satisfactory perfomance testing

GradientBoostingClassifier(learning_rate=0.01, max_depth=2, max_features='auto',
                           random_state=0)
LogisticRegression(C=0.01, max_iter=10, random_state=0, tol=0.01)
MLPClassifier(batch_size=128, max_iter=500, random_state=0)
OneVsRestClassifier(estimator=LinearSVC(), n_jobs=1)
GaussianNB(var_smoothing=1e-07)
LinearSVC(C=0.0001, max_iter=10, random_state=0, tol=0.01)
DecisionTreeClassifier(max_depth=2, random_state=0)
RandomForestClassifier(max_depth=10, random_state=0)
Performance testing successful
```

Figure 53: Invariant, directional expectation, and performance test results

All three tests for sklearn models work as expected. That is changing irrelevant features should not lead to accuracy improvement and we should expect a change in accuracy when we change relevant features.

(c) Precision Recall Curve of Optimized models

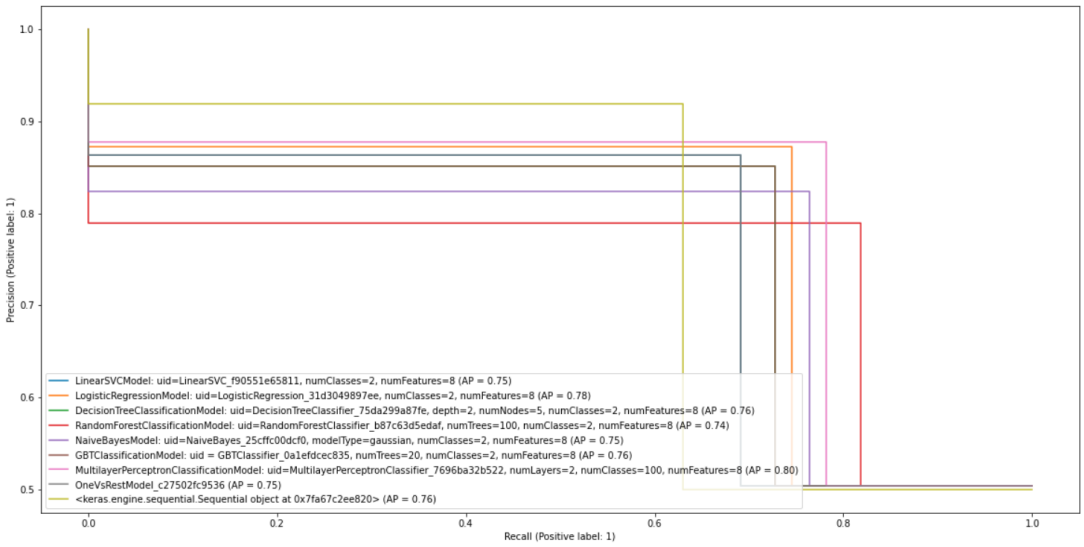

Figure 54: Precision Recall Curve of pyspark models

The curves show the precision-recall tradeoff at different thresholds. A high area under the curve represents both high recall and high precision. High precision relates to a low FPR, and high recall relates to a low FNR.

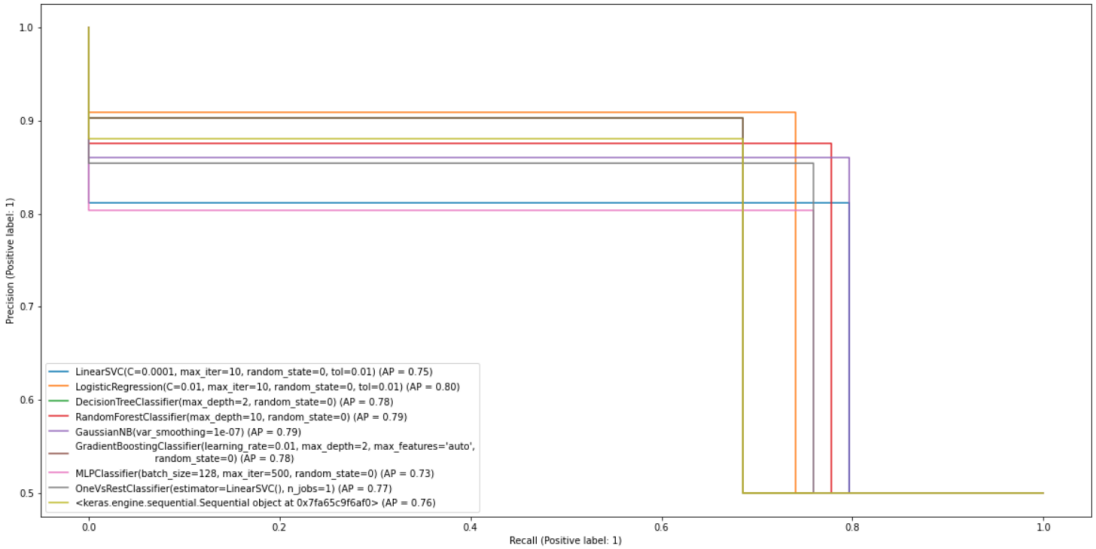

Figure 55: Precision Recall Curve of sklearn models

(d) Accuracy of Optimized Models for Diabetes

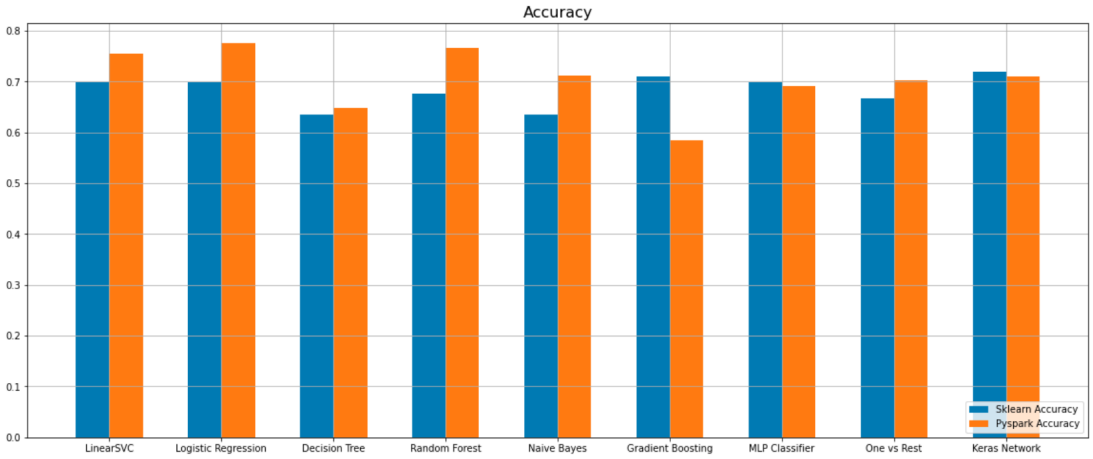

Figure 56: Accuracy of optimized models for diabetes datasetet

Compared to the titanic optimized models the accuracy is less for many of the models.

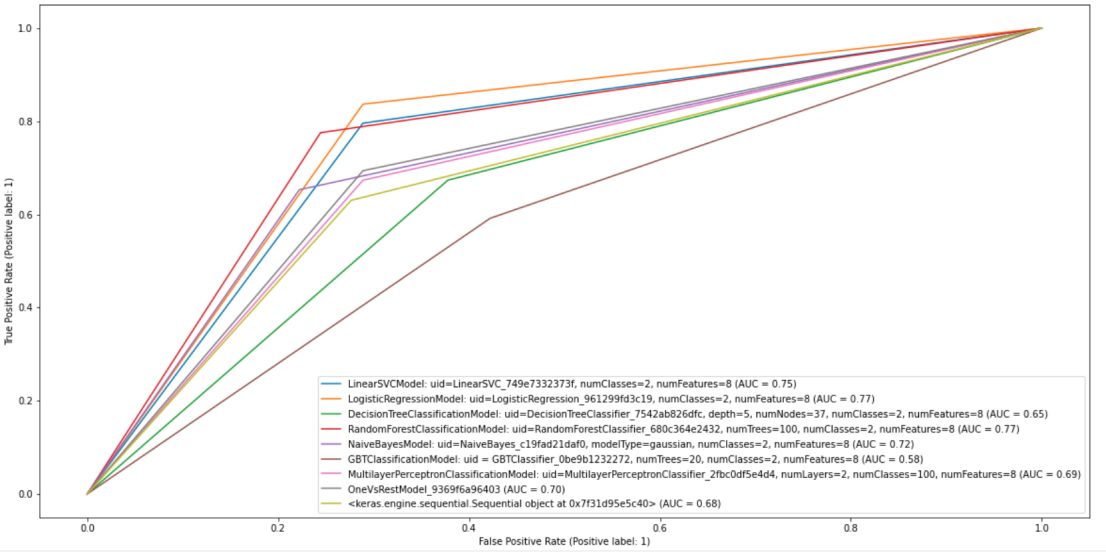

Figure 57: RoC curve of optimized pyspark models for diabetes datasets